Relatório Final – Iniciação Científica
Estágio científico e tecnológico II (EE016)

Estudo e Aplicação de Modelos de Previsão no Contexto de Séries de Vazões de Rios

Submetido à
Faculdade de Engenharia Elétrica e de Computação (FEEC)

Departamento de Engenharia de Computação e Automação Industrial (DCA)
Faculdade de Engenharia Elétrica e de Computação (FEEC)
Universidade Estadual de Campinas (UNICAMP)
CEP 13083-852, Campinas, São Paulo (SP)

Candidato: Daniel da Costa Nunes Resende Neto
Orientador: Prof. Levy Boccato

1 Introdução

Uma série temporal é uma sequência de medidas feitas ao longo do tempo sobre um fenômeno de interesse. Em várias áreas do conhecimento, tais como engenharia, economia, matemática aplicada e computação, diversas séries temporais são particularmente relevantes não só pela perspectiva de análise e interpretação de seus históricos passados, mas também pela possibilidade de estimar seus valores futuros. Este último desafio dá origem ao problema de predição de séries temporais [1].

Um modelo genérico de previsão pode ser visto na Figura 1.

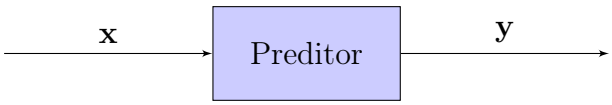

Figura 1: Estrutura geral de um preditor.

O preditor recebe um vetor $\mathbf{x} = [\, x(n-L) \quad x(n-L-1) \quad \ldots \quad x(n-M-L+1)\,]^T$ de M amostras atrasadas da série temporal e produz uma saída $\mathbf{y}$, sendo L o passo de predição.

Na primeira etapa deste projeto foram estudados dois modelos de predição clássicos da literatura, ambos de natureza linear: auto-regressivo (AR) e auto-regressivo de médias móveis (ARMA) [1], detalhados nas Seções 2.1 e 2.3 do relatório parcial.

Ambos foram implementados, com base em elementos da metodologia Box-Jenkins [1], e estudados no contexto da série de vazões de Água Vermelha (ver Seções 4.1-4.4 do relatório parcial). No caso do modelo ARMA, exploramos uma meta-heurística denominada *clearing* [2, 3] para o ajuste de seus parâmetros, visando minimizar o erro quadrático médio na etapa de treinamento (vide Seção 3 do relatório parcial).

Por fim, ambos os modelos foram comparados a partir do Erro Quadrático Médio (EQM) gerado para os dados de teste da série de Água Vermelha. Os desempenhos atingidos foram satisfatórios e relativamente próximos: enquanto o AR alcançou um EQM de 1088,6 $(\mathrm{m}^3/\mathrm{s})^2$, o ARMA obteve um EQM de 1056,3 $(\mathrm{m}^3/\mathrm{s})^2$, conforme apresentado na Seção 4.5 do relatório parcial.

Para a segunda parte do projeto (como mencionado na Seção 5 do relatório parcial), incorpora-se a implementação dos modelos AR e ARMA, considerando agora a série de vazões de Jirau (ver Seção 3.5) durante o período idêntico ao da série de Água Vermelha. Também, incorpora-se o estudo (ver Seção 2.2) e a implementação (ver Seções 3.2 a 3.4) de métodos não-lineares baseados em redes neurais [4] para a previsão das séries de Água Vermelha e de Jirau. São eles: a rede *perceptron* de múltiplas camadas (MLP, do inglês

multilayer perceptron), como representante da classe de modelos *feedforward*; assim como três modelos recorrentes: as RNNs (do inglês, *recurrent neural networks*) simples, as LSTM (do inglês, *long short-term memory*) [5] e as GRUs (do inglês, *gated recurrent units*) [6].

Então, foi realizada uma análise comparativa entre estes seis modelos no contexto das duas séries de vazões escolhidas, tendo como base o EQM para o conjunto de teste. A metodologia completa, os resultados obtidos e as conclusões finais do projeto encontram-se nas Seções 3.6 e 4 deste documento.

2 Modelos de Previsão

2.1 Modelos lineares

A exposição detalhada sobre os modelos AR e ARMA encontra-se na Seção 2 do relatório parcial. Não obstante, é pertinente recordar as equações que resumem o comportamento entrada-saída destes modelos.

O modelo AR produz uma estimativa do valor futuro da série através de uma combinação linear de alguns valores passados, conforme mostra a expressão abaixo:

$$x(n) = a_1 x(n-L) + \ldots + a_M x(n-M-L+1) + \eta(n), \tag{1}$$

onde M determina a ordem do modelo, $a_i, i = 1, \ldots, M$ são os coeficientes que ponderam as amostras passadas da série e $\eta(n)$ denota o erro instantâneo, o qual possui uma distribuição normal cuja média é nula e cuja variância (σ_η^2) é constante (também chamado ruído branco).

Por sua vez, o modelo ARMA explora não só os valores passados da própria série, como também amostras passadas do erro cometido na previsão. Assim, a estimativa é gerada da seguinte forma:

$$x(n) = \sum_{k=0}^{M} a_k x(n-k-L+1) + \sum_{k=0}^{K} b_k \eta(n-k-J+1), \tag{2}$$

onde $b_0 = 1$, J é o passo do ruído branco, K representa a ordem do modelo e as outras definições ainda valem.

Passaremos, agora, à apresentação dos fundamentos teóricos dos modelos não-lineares utilizados no trabalho.

2.2 Modelos não-lineares

Os modelos não-lineares estudados pertencem todos à classe de redes neurais artificiais (ANNs, do inglês *artificial neural networks*). Em síntese, as ANNs se inspiram em princípios ligados ao funcionamento do sistema nervoso, em particular, do cérebro, sendo formadas por várias unidades simples de processamento, denominadas neurônios artificiais, que desempenham um papel análogo ao do neurônio biológico.

Um modelo tipicamente explorado em ANNs para o neurônio artificial [4] é mostrado na Figura 2.

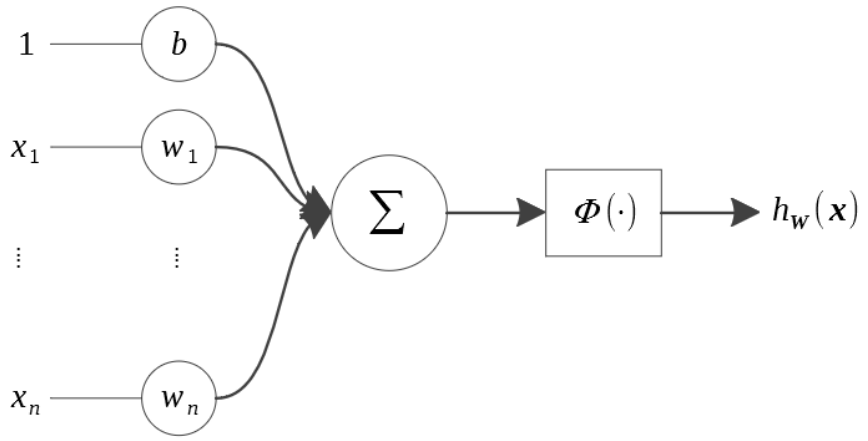

Figura 2: Modelo de neurônio artificial do tipo *perceptron*.

Neste modelo, os atributos x_i contidos na instância $\mathbf{x}$ e a saída $h_{\mathbf{w}}(\mathbf{x})$ são números reais, sendo que cada elemento de entrada x_i está associado a um peso $w_i \subset \mathbf{w}$. Aplica-se, então, uma função de ativação ϕ (discutida em breve) sobre a soma ponderada de suas entradas mais um termo de bias (b) de forma a gerar a saída:

$$h_{\mathbf{w}}(\mathbf{x}) = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{3}$$

A função de ativação ϕ pode assumir várias formas, sendo as mais utilizadas as funções: logística (também chamada, em inglês, *sigmoid*), tangente hiperbólica (Tanh), ReLu e SeLu. Estas funções e suas respectivas derivadas (utilizadas durante o treinamento da rede, discutido na próxima seção) estão representadas na Figura 3.

É de se notar que, se nenhuma função de ativação fosse aplicada ao ligar-se um neurônio artificial ao outro, estaríamos encadeando somente transformações lineares, o que no fim das contas é uma única transformação linear. A falta de não-linearidade neste caso impediria a resolução de problemas mais complexos, já que o encadeamento de neurônios (ou ainda camadas de neurônios) seria, para todos os efeitos, um único neurônio.

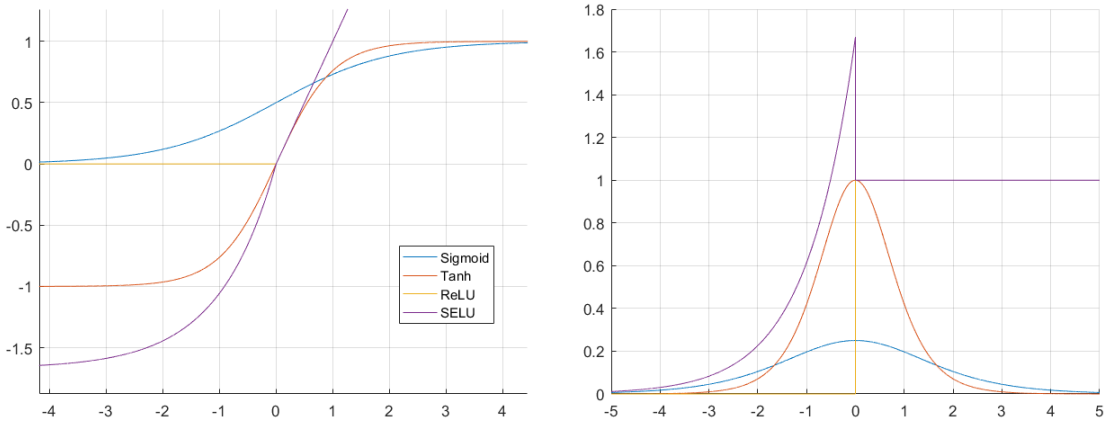

Figura 3: Funções de ativação (à esquerda) e suas derivadas (à direita).

Em algumas arquiteturas de redes recorrentes, o modelo de neurônio pode apresentar algumas diferenças importantes. Desde modo, deixaremos sua descrição para a Seção 2.2.2, a qual introduz os detalhes deste tipo de rede.

2.2.1 MLP

As MLPs são redes neurais formadas fundalmentamente por múltiplas camadas de neurônios do tipo *perceptron*. A Figura 4 apresenta a estrutura geral de uma MLP.

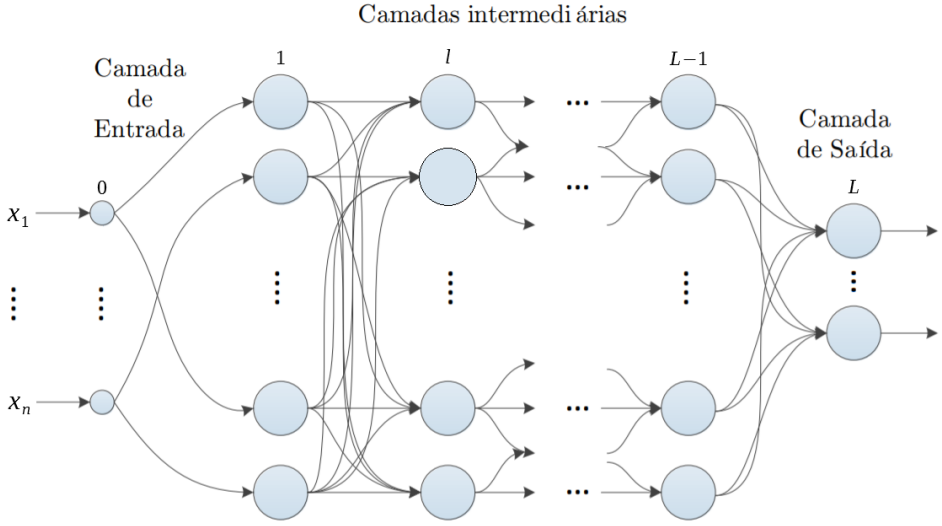

Figura 4: Arquitetura de uma MLP. Figura extraída de [7].

Esta rede é composta, primeiramente, por uma camada ($l = 0$) de entrada. Os neurônios desta camada apenas passam para suas saídas o atributo com o qual foram alimentados.

As próximas camadas, exceto pela última, são denominadas camadas intermediárias (em inglês, *hidden layers*). Em cada camada, os estímulos de entrada são combinados linearmente, com base nos pesos de cada neurônio, e a função de ativação é aplicada sobre o resultado acumulado. Assim, as saídas dos neurônios de uma camada são propagadas adiante para a próxima camada, até que cheguem à última camada da rede.

Uma vez que todos os neurônios da camada l estão conectados a todos os neurônios da próxima camada $(l+1)$, diz-se que estas camadas são totalmente conectadas (em inglês, *fully connected*), ou densas (em inglês, *dense*).

Por fim, a camada de saída (em inglês, *output layer*) é quem gera as respostas da rede para uma instância. Curiosamente, o tipo de estrutura empregado na saída de uma rede depende da tarefa considerada. Por exemplo, em um problema de regressão, a camada de saída pode ser simplesmente um combinador linear (ou seja, uma camada densa com função de ativação dada pela identidade). Já no caso de classificação, é comum o uso de um neurônio com função logística (para o caso binário), ou, então, a função *softmax*, a qual gera uma saída para cada classe possível.

A equação (3) pode ser generalizada matricialmente (ver equações em (4)) para definir a matriz contendo os vetores de saídas de uma camada gerados a partir de uma sequência de vetores de entrada. Isto é feito separadamente para a camada de entrada (equação (4a)), uma camada intermediária (equação (4b)) e a camada de saída (equação (4c)). Essa representação possibilita o cálculo da matriz de saída de uma rede alimentada com um subconjunto de instâncias chamado *mini-batch*. Ela é utilizada no treinamento da rede (abordado em breve).

$$\mathbf{H}^0(\mathbf{X}) = \mathbf{X}, \tag{4a}$$

$$\mathbf{H}^l_{\mathbf{W^l},\mathbf{B^l}}(\mathbf{H}^{l-1}) = \phi(\mathbf{H}^{l-1}\mathbf{W}^l + \mathbf{B}^l), \tag{4b}$$

$$\mathbf{H}^L_{\mathbf{W^L},\mathbf{B^L}}(\mathbf{H}^{L-1}) = \phi(\mathbf{H}^{L-1}\mathbf{W}^L + \mathbf{B}^L) \tag{4c}$$

Nestas equações:

- A matriz $\mathbf{X} = [\mathbf{x}_1\ \mathbf{x}_2\ \ldots\ \mathbf{x}_d]^\top$, com $\mathbf{X} \in \mathbb{R}^{d\times n}$, é o *mini-batch* recebido pela rede e contém d instâncias $\mathbf{x}_i^\top$ em suas linhas, sendo que cada instância é caracterizada por n atributos;
- Os vetores coluna $\mathbf{w}_j^l$ da matriz $\mathbf{W}^l = [\mathbf{w}_1^l\ \mathbf{w}_2^l\ \ldots\ \mathbf{w}_{m(l)}^l]$, com $\mathbf{W}^l \in \mathbb{R}^{m(l-1)\times m(l)}$, contêm os conjuntos de pesos pertencentes a cada um dos $m(l)$ neurônios da camada

l. Note que o número de pesos de um neurônio da camada l (a dimensão dos vetores $\mathbf{w}_j^l$) é igual ao número de neurônios da camada anterior $m(l-1)$, já que este é também seu número de saídas. Os valores assumidos por $m(l)$ quando $l = 1, ..., L$ são definidos na construção da rede e, como as saídas da camada de entrada são as próprias entradas, $m(0) = n$;

- A matriz $\mathbf{H}^l = [\mathbf{y}_1^l \ \mathbf{y}_2^l \ \dots \ \mathbf{y}_d^l]^\top$, com $\mathbf{H}^l \in \mathbb{R}^{d \times m(l)}$, contém os d vetores linha de saídas $\mathbf{y}_i^{l\top}$ da camada l. Cada um deles contém $m(l)$ saídas;

- Os vetores linha $\mathbf{b}_i^{l\top}$ da matriz $\mathbf{B}^l = [\mathbf{b}_1^l \ \mathbf{b}_2^l \ \dots \ \mathbf{b}_d^l]^\top$, com $\mathbf{B}^l \in \mathbb{R}^{d \times m(l)}$, são idênticos entre si e contêm os $m(l)$ valores de bias de cada neurônio da camada l;

- A função de ativação ϕ é aplicada a todos os elementos da matriz $(\mathbf{H}^{l-1}\mathbf{W}^l + \mathbf{B}^l)_{d \times m(l)}$, a qual contém em seu elemento (i, j) a soma dos elementos do i-ésimo (de um total de d) vetor de saída da camada anterior ponderados pelos pesos do j-ésimo neurônio da camada l, mais o termo de bias.

O objetivo agora é treinar a rede. Em outras palavras, encontrar os valores de todos os pesos sinápticos, $\mathbf{W}^l, l = 1, \dots, L$, que minimizem uma medida de erro entre as saídas geradas pela rede e as saídas desejadas.

No contexto de regressão/predição, um critério classicamente explorado é o EQM. O processo de treinamento dá origem, então, a um problema de otimização não-linear irrestrito. Neste contexto, o uso de algoritmos iterativos baseados em gradiente constitui a opção padrão na literatura, inclusive para modelos profundos [8]. Para isso, é preciso computar a derivada da função custo (EQM) com respeito a todos os pesos da rede, o que pode ser feito com o auxílio do algoritmo de retropropagação do erro (em inglês, *error backpropagation*) [9].

Segue, de forma geral, a descrição deste algoritmo:

I. Primeiramente, é importante destacar que o algoritmo lida com um *mini-batch* $\mathbf{X}$, contendo d instâncias por vez. Além disso, ele passa por todo o conjunto de treinamento $\mathbf{X}_{\text{train}}$ múltiplas vezes, e cada passagem é chamada de época (do inglês, *epoch*).

II. Então, cada *mini-batch* $\mathbf{X}$ é passado para a camada de entrada ($l = 0$) da rede, a qual o passa para a primeira camada intermediária ($l = 1$) que computa todas as suas saídas a partir da equação (4b). As saídas desta camada são passadas como entrada para a próxima camada (utilizando-se novamente a equação (4b)), agora

com $l = 2$. Seguindo esta lógica, calculam-se todas as saídas de todas as camadas da MLP, inclusive a última pela equação (4c). Este processo caracteriza a etapa de propagação adiante (do inglês, *forward pass*).

III. O próximo passo é utilizar uma função de perda (do inglês, *loss function*) para comparar os vetores de saídas desejadas (no caso de um *mini- batch* de tamanho $d > 1$) com os vetores de saídas produzidos pela rede. Esta função, tipicamente o EQM, retorna alguma medida do erro cometido.

IV. A partir da camada de saída, calculamos o gradiente do erro com respeito a todos os pesos da rede. Para isso, é preciso explorar a regra da cadeia para propagar as derivadas até a camada de entrada. Este passo recebe o nome de retropropagação do erro, pois voltamos através da estrutura da rede, no sentido contrário ao fluxo de dados, para computar a derivada do erro em relação aos pesos. No Apêndice A deste relatório, apresentamos com mais detalhes a fundamentação matemática do *backpropagation*.

V. Com todos os gradientes computados a partir dos atributos em $\mathbf{X}$, o valor do gradiente de fato utilizado para atualizar um peso da rede é a média dos gradientes (referentes àquele peso) calculados para aquele *mini-batch*. Isto garante que cada peso seja atualizado a cada d instâncias consideradas, o que acelera o algoritmo em detrimento da precisão. Este processo se repete para todos os *mini-batches* ao longo de várias épocas.

O otimizador clássico utilizado na atualização dos pesos da rede é o gradiente descendente [4]. Como o gradiente de uma função aponta para sua direção de maior crescimento, o método se aproveita da direção oposta do gradiente para minimizar (daí o nome) a função de perda.

Assim, a atualização de um peso arbitrário w_a da rede é feita através da subtração do gradiente da função de perda em relação a este peso, da seguinte forma:

$$w_a^{(\text{próximo passo})} = w_a - \eta \frac{\partial E}{\partial w} \tag{5}$$

onde η é o passo (ou taxa) de aprendizado (em inglês, *learning rate*) e E é a função de perda.

Contudo, a otimização por gradiente descendente sofre comumente com o desaparecimento dos gradientes (em inglês, *vanishing gradients problem*), ou ainda o contrário: o seu aumento desenfreado (em inglês, *exploding gradients problem*), que é

mais comum no contexto de redes recorrentes. Isso acontece à medida que os gradientes de erro são propagados para as camadas inferiores da rede, especialmente se esta for densa.

Uma técnica desenvolvida para tentar evitar ambos os problemas supracitados é a normalização do *batch* (BN, do inglês *batch normalization*) [4]. Ela consiste na adição de uma operação dentro do modelo logo antes ou depois da função de ativação de cada camada intermediária.

Esta operação simplesmente centra todas as instâncias ao redor do zero e as normaliza. Para isso, o algoritmo deve estimar a média e o desvio padrão de cada atributo dentro do *mini-batch* (daí o nome) que alimenta a camada de *batch normalization*. Isso é feito da seguinte maneira:

$$\boldsymbol{\mu}_B = \frac{1}{d}\sum_{i=1}^{d}\mathbf{x}_i \tag{6a}$$

$$\boldsymbol{\sigma}_B^2 = \frac{1}{d}\sum_{i=1}^{d}(\mathbf{x}_i - \boldsymbol{\mu}_B)^2 \tag{6b}$$

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}} \tag{6c}$$

$$\mathbf{z}_i = \boldsymbol{\gamma} \otimes \hat{\mathbf{x}}_i + \boldsymbol{\beta} \tag{6d}$$

onde $\boldsymbol{\mu}_B$ e $\boldsymbol{\sigma}_B$ são os vetores contendo, respectivamente, a média e o desvio padrão de cada atributo para o *mini-batch*; $\hat{\mathbf{x}}_i$ é a i-ésima instância centrada em zero e normalizada; $\boldsymbol{\gamma}$ é o vetor que pondera cada um dos atributos normalizados em $\hat{\mathbf{x}}_i$; $\otimes$ representa o produto elemento a elemento; o vetor $\boldsymbol{\beta}$ contém o *offset* relativo a cada atributo normalizado; ϵ é um número pequeno que impede a divisão por zero (tipicamente 10^{-5}); e $\mathbf{z}_i$ é a saída reescalada e deslocada da camada de *batch normalization*.

O uso de BN introduz dois novos parâmetros por camada: um que escala os atributos normalizados $\boldsymbol{\gamma}$ e outro que os desloca $\boldsymbol{\beta}$. Cabe ao modelo aprender a escala e o deslocamento ótimo de cada atributo para cada camada.

Isso é feito com eficiência durante o treinamento, pois os *mini-batches* são muitos e de tamanho significativo. O mesmo não vale para os dados de validação, pois ainda que as previsões não sejam padrões isolados e pertençam a um *batch*, este costuma ser pequeno e pouco numeroso (devido ao menor número de dados). Desta forma, realizar o *batch normalization* diretamente sobre os dados de validação não é confiável.

O que se faz nesse caso é criar outros dois vetores de parâmetros ($\boldsymbol{\mu}'_B$ e $\boldsymbol{\sigma}'_B$), contendo as médias e desvios padrões (respectivamente) finais de cada atributo das camadas de *batch normalization*. Estes valores finais são o resultado do método de médias móveis aplicado

sobre as médias e os desvios padrões de todas as instâncias durante o treinamento. Assim, o operador *batch normalization* utiliza apenas as equações (6c) e (6d) com $\boldsymbol{\mu}_B = \boldsymbol{\mu}'_B$ e $\boldsymbol{\sigma}_B = \boldsymbol{\sigma}'_B$ para todas as instâncias de validação.

Outro problema característico do gradiente descendente é sua lentidão, a qual pode ser contornada com a utilização de outros otimizadores. Estes continuam dependendo somente das derivadas parciais de primeira ordem (Jacobianos).

O otimizador explorado neste trabalho é o Nadam (do inglês, *Nesterov adaptive moment estimation*). O conjunto de equações que o descreve é dado por:

$$\mathbf{w}' = \mathbf{w} + \beta_3 \mathbf{m} \tag{7a}$$

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\mathbf{w}} E(\mathbf{w}') \tag{7b}$$

$$\mathbf{s} \leftarrow \beta_2 \mathbf{s} - (1 - \beta_2) \nabla_{\mathbf{w}} E(\mathbf{w}') \otimes \nabla_{\mathbf{w}} E(\mathbf{w}') \tag{7c}$$

$$\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1} \tag{7d}$$

$$\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2} \tag{7e}$$

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \epsilon} \tag{7f}$$

No método clássico do gradiente descendente, o peso $\mathbf{w}$ é atualizado diretamente subtraindo-se o gradiente da função custo relativo a ele naquele instante (ver equação 5).

Já a otimização por momento presente no Nadam incorpora a influência de gradientes passados, não apenas o local. Isto é feito subtraindo-se o gradiente local, desta vez, do vetor de momento $\mathbf{m}$ (ver equação (7b)). E este, por sua vez, é utilizado para atualizar o peso $\mathbf{w}$ (veja a equação (7f)).

O gradiente passa, então, a transmitir a ideia de aceleração, e não mais velocidade. Assim, o algoritmo acumula velocidade de convergência caso esteja caminhando para um ótimo local e escapa de platôs mais rapidamente.

O hiperparâmetro β_1 (ver equação (7b)) é utilizado para prevenir que o momento cresça desenfreadamente e para que o valor do momento $\mathbf{m}$ se torne, a partir da multiplicação do termo $(1 - \beta_1)$, uma média exponencialmente decrescente de valores passados. Este hiperparâmetro é tipicamente inicializado em 0,9.

Para que o algoritmo não avance velozmente em uma direção que não aponta para o ótimo local (como acontece no metódo clássico do gradiente descendente), introduz-se o vetor $\mathbf{s}$ (ver equação (7c)), o qual escala o vetor momento (ver equação (7f)).

Esta técnica causa, no geral, uma diminuição no gradiente (devido ao termo ao quadrado na definição do vetor $\mathbf{s}$), especialmente se este toma grandes proporções. Assim, é possível corrigir a sua direção para um ponto mais próximo do ótimo local antes de uma convergência acelerada do algoritmo.

Para que esta redução do gradiente não cause lentidão significativa na convergência do algoritmo, introduz-se um decaimento exponencial na consideração de valores passados de $\mathbf{s}$ durante sua atualização (ver equação (7c)), através dos termos β_2 e $1 - \beta_2$. Este hiperparâmetro é tipicamente inicializado em $0,999$ e ϵ em 10^{-7}, evitando a divisão por 0.

Como $\mathbf{m}$ e $\mathbf{s}$ são inicializados em 0, ambos tenderiam a 0 no começo do treinamento. As equações (7d) e (7e) visam impulsionar a atualização de $\mathbf{m}$ e $\mathbf{s}$ neste começo, aproveitando os valores iniciais próximos de 1 de β_1 e β_2.

Por fim, o Nadam também incorpora a ideia proposta pelo matemático Yurii Nesterov (daí o nome) de que o gradiente deve ser medido não sobre o ponto atual $\mathbf{w}$, mas sobre um ponto levemente à frente na direção do vetor momento, como faz a equação (7a). Esta ideia pode ser benéfica, uma vez que a direção deste aponta geralmente na direção correta (a direção do ponto ótimo), tornando a atualização dos pesos mais precisa.

Com isso, concluímos o estudo teórico da rede *feedforward* MLP e dos elementos básicos de uma rede neural (como a função de ativação e o otimizador), os quais serviram de base para este projeto. Nosso foco se direcionará, nas próximas seções, às redes neurais do tipo recorrente.

2.2.2 RNN

Uma RNN apresenta uma estrutura bastante similar à de uma MLP, mas com a presença de retroalimentações. A célula básica de uma RNN consiste em um único neurônio recorrente, o qual recebe entradas (mais um termo bias) e produz uma saída. Esta é alimentada à sua própria entrada, como mostra a Figura 5 (esquerda).

A cada instante de tempo t (também chamado de *frame*), este neurônio recebe o vetor de entradas $\mathbf{x}_{(t)}$ assim como sua própria saída do instante de tempo anterior, $y_{(t-1)}$. Como não há saída prévia no instante 0, o primeiro valor de y é nulo.

Para facilitar o entendimento, representamos esta rede (composta por um neurônio) através do tempo, como mostra a Figura 5 (direita). Isto é chamado de desenrolamento da rede no tempo, pois o mesmo neurônio é representado várias vezes, conforme o número de instantes de tempo considerados.

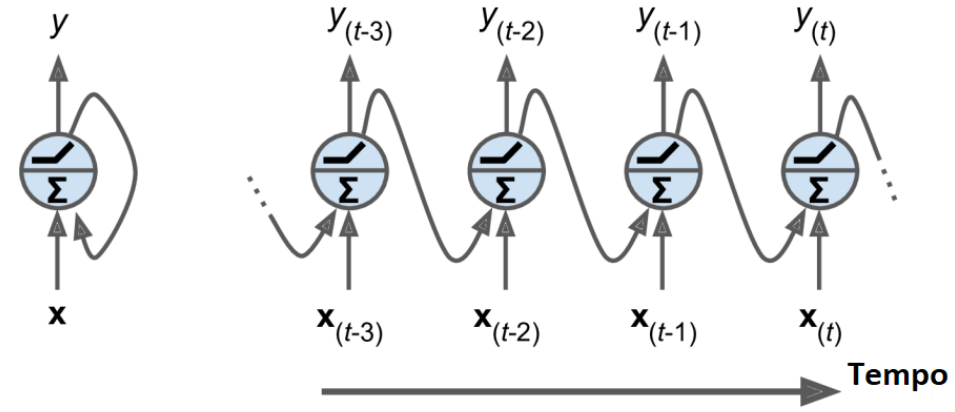

Figura 5: Neurônio recorrente representado através do tempo. Figura extraída de [4].

A criação de uma camada de neurônios recorrentes é intuitiva. Cada um dos neurônios continua recebendo o vetor de entradas $\mathbf{x}_{(t)}$, porém passa a receber agora um vetor de saídas $\mathbf{y}_{(t-1)}$, o qual contém as saídas passadas dos neurônios desta camada. Esta estrutura pode ser vista na Figura 6.

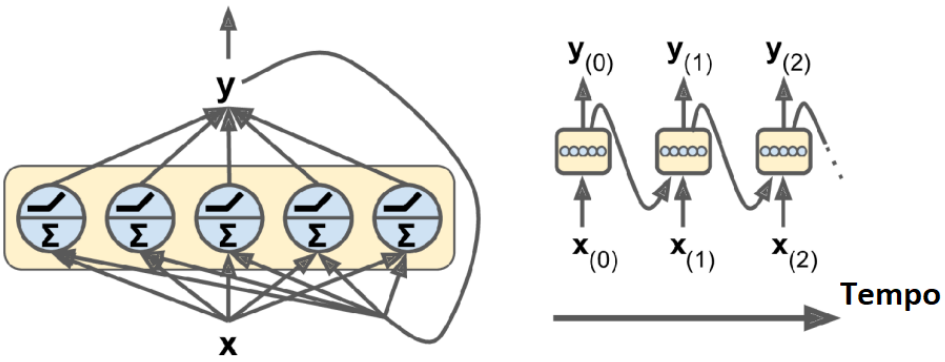

Figura 6: Camada de neurônios recorrentes. Figura extraída de [4].

Sendo assim, cada neurônio recorrente possui dois vetores de pesos: o primeiro, $\mathbf{w}_x$, pondera as entradas instantâneas $\mathbf{x}_{(t)}$; o segundo, $\mathbf{w}_y$, é responsável por ponderar as saídas passadas, $\mathbf{y}_{(t-1)}$. Considerando agora uma camada recorrente, podemos agrupar todos os pesos em duas matrizes, $\mathbf{W}_x$ e $\mathbf{W}_y$, de maneira que o vetor de saída da camada pode ser escrito como:

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{x}_{(t)}^{\top}\mathbf{W}_x + \mathbf{y}_{(t-1)}^{\top}\mathbf{W}_y + \mathbf{b}^{\top}\right)^{\top} \tag{8}$$

sendo $\mathbf{b}$ o vetor de bias.

De modo análogo ao que foi feito para a MLP, podemos agrupar várias instâncias em um único *mini-batch* $\mathbf{X}_{(t)}$, de modo que a matriz de saídas é dada por:

$$\begin{aligned}\mathbf{Y}_{(t)} &= \phi\left(\mathbf{X}_{(t)}\mathbf{W}_x + \mathbf{Y}_{(t-1)}\mathbf{W}_y + \mathbf{B}\right) \\ &= \phi\left(\begin{bmatrix}\mathbf{X}_{(t)} & \mathbf{Y}_{(t-1)}\end{bmatrix}\mathbf{W} + \mathbf{B}\right) \text{ com } \mathbf{W} = \begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix}\end{aligned} \tag{9}$$

onde a matriz $\mathbf{Y}_{(t)}$ contém em cada linha as saídas de uma instância do *mini-batch*; a matriz $\mathbf{X}_{(t)}$ contém em cada linha uma diferente instância de tamanho igual à quantidade de atributos; as matrizes $\mathbf{W}_x$ e $\mathbf{W}_y$ contêm em cada coluna os pesos que multiplicam os atributos e as saídas prévias de uma instância, respectivamente; a matriz $\mathbf{B}$ contém o mesmo vetor de bias de cada neurônio em cada uma de suas linhas; e as matrizes $\mathbf{W}_x$ e $\mathbf{W}_y$ podem ser concatenadas em uma única matriz $\mathbf{W}$, como mostrado.

Devido à relação de recorrência presente em (9), $\mathbf{Y}_{(t)}$ acaba sendo influenciado por todas as entradas desde $t = 0$, *i.e.*, $\mathbf{X}_{(0)}, \mathbf{X}_{(1)}, \ldots, \mathbf{X}_{(t)}$. Por isso, diz-se que a rede recorrente possui memória. A parte da rede que preserva estados ao longo do tempo é chamada de célula de memória (ou simplesmente célula).

A estrutura aqui apresentada (uma RNN simples) é uma célula muito básica, capaz apenas de aprender padrões curtos. As LSTMs e GRUs, apresentadas nas seções 2.2.3 e 2.2.4, buscam justamente a expansão desta memória.

Estas estruturas também realizam a retroalimentação a partir das próprias saídas y de cada neurônio. Contudo, é comum que se utilize uma função $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$ para representar o estado da célula atual e, consequentemente, para a retroalimentação. Já sua saída no instante t seria dada por $\mathbf{y}_{(t)} = g(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$, também uma função da entrada atual e do estado anterior.

No caso de células básicas, $f = g$, fazendo com que a saída $\mathbf{y}_{(t)}$ e o estado atual $\mathbf{h}_{(t)}$ sejam idênticos. Porém, em células mais complexas, como as representadas na Figura 7, isto nem sempre acontece.

Baseando-se nesta representação, uma RNN pode ser alimentada com uma sequência de entradas (isto é, um vetor de entradas por instante) e gerar uma sequência de saídas, como mostra o esquema no canto superior-esquerdo da Figura 8.

Este tipo de rede é chamado sequência-sequência e pode ser utilizado no contexto de séries temporais se considerarmos, por exemplo, um caso simples em que $\mathbf{X}_{(t)}$ é o valor de uma série temporal no instante de tempo t, sendo a série de tamanho N. Já a saída $\mathbf{Y}_{(t)}$ é a estimativa da RNN para o valor da série no instante de tempo seguinte $(t + 1)$. Deste modo, podemos alimentar a RNN com os valores desta série um a um, obtendo como saída as predições de seus valores do instante 2 ao $N + 1$.

A saída $\mathbf{Y}_{(t)}$ poderia conter, alternativamente, as estimativas da RNN para os m

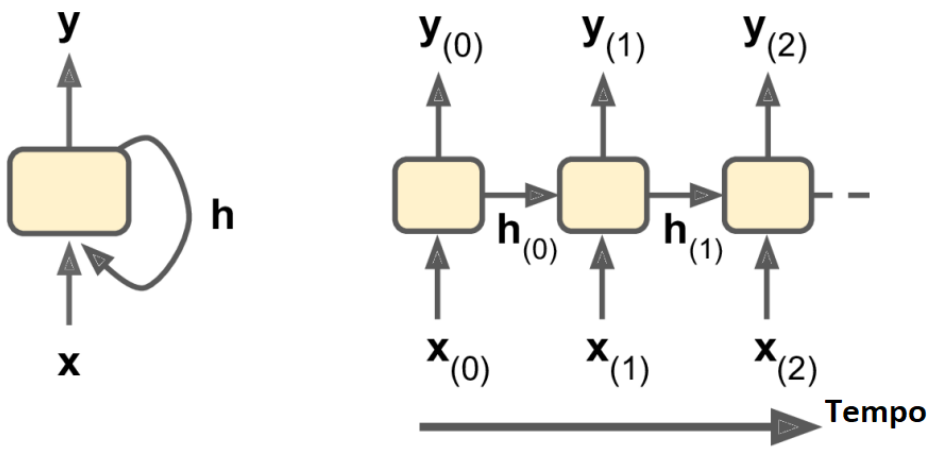

Figura 7: A diferença entre o estado de uma célula e sua saída. Figura extraída de [4].

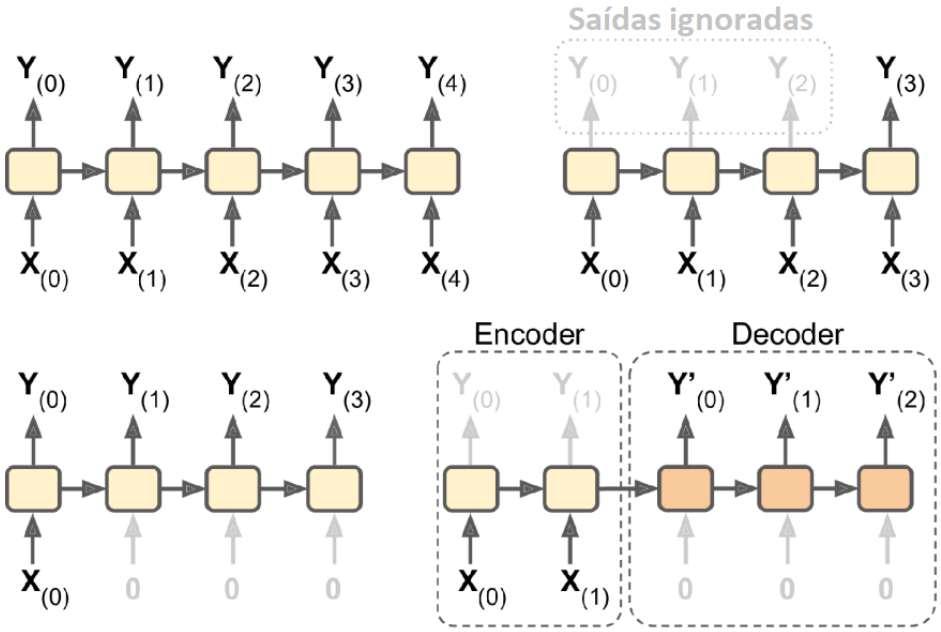

Figura 8: Modos de implementação de uma RNN. Figura extraída de [4].

próximos valores da série. Assim, é possível predizer os valores da série do instante 2 ao $N + m$.

Por outro lado, a rede pode ser alimentada com uma sequência de entradas e produzir uma única saída de interesse no último instante de tempo, como mostra o diagrama no canto superior-direito da Figura 8. Esta é uma rede sequência-vetor.

No contexto de séries temporais, este tipo de rede consegue prever, por exemplo, os próximos m valores (desconhecidos) de uma série. Contudo, é possível dividir uma única série temporal em partes menores (deslocadas de um instante de tempo) de modo que a rede sequência-vetor produza como um todo os $N-1$ valores da série mais os m próximos, assim como o modelo sequência-sequência. Isto foi feito na segunda parte deste trabalho

e está detalhado na Seção 3.2.

Resumidamente, a predição de séries temporais pode ser feita de um jeito ou de outro, dependendo de como os dados da série são divididos.

Por outro lado, a rede pode ser do tipo vetor-sequência, como mostra o esquema no canto inferior-esquerdo da Figura 8. A aplicação mais comum neste caso é a classificação de imagens.

É possível também utilizar um modelo sequência-vetor seguido por um vetor-sequência, chamado *Encoder-Decoder*. Ele é muito utilizado, por exemplo, para traduzir sentenças. O codificador recebe uma frase por inteiro (pois a tradução deve ser feita conhecendo-se a frase por completo) e gera um vetor único de saída. Este, por sua vez, é decodificado em uma frase em outra língua. Este último processo está representado no canto inferior-direito da Figura 8.

Uma RNN é treinada utilizando-se a técnica de retropropagação do erro através do tempo (do inglês, *backpropagation through time*). Para este fim, aplica-se a retropropagação do erro (analogamente ao que foi feito para a MLP) à RNN representada através de n instantes de tempo (veja a Figura 9).

Assim como na retropropagação regular, o primeiro passo consiste em gerar todas as saídas $\mathbf{Y}$ de todos os instantes de tempo (*forward pass*). Esse é representado pelas setas em pontilhado na Figura 9.

A função de perda E é, então, aplicada sobre todas as saídas relevantes (no caso da Figura 9, as 3 últimas). Os gradientes desta função em relação aos pesos das matrizes $\mathbf{W}$ e $\mathbf{B}$ são calculados e propagados em direção aos menores instantes de tempo (representado pelas setas sólidas). Contudo, esta propagação só acontece para os instantes de tempo cuja saída é considerada relevante (no exemplo, os instantes 2, 3 e 4).

Sabendo que os pesos em $\mathbf{W}$ e $\mathbf{B}$ são os mesmos para todo instante de tempo (pois a rede é única), o gradiente em relação a um peso é, na verdade, a soma de todos os gradientes calculados para todos os instantes de tempo relevantes àquele peso. Para a atualização de um peso, o gradiente utilizado é, em última análise, a média entre os gradientes com respeito àquele peso gerados para o *mini-batch* em questão.

É importante destacar que a retropropagação do erro ao longo do tempo faz surgir potências da matriz de pesos $\mathbf{W}$ na expressão do gradiente. A magnitude destas potências aumenta com o número de instantes considerados relevantes, ou seja, instantes pelos quais se propagou o erro. Caso essa magnitude seja grande, um autovalor para a matriz $\mathbf{W}$ maior do que 1 torna o gradiente exorbitante; já um autovalor menor do que 1 faz com que esse gradiente desapareça.

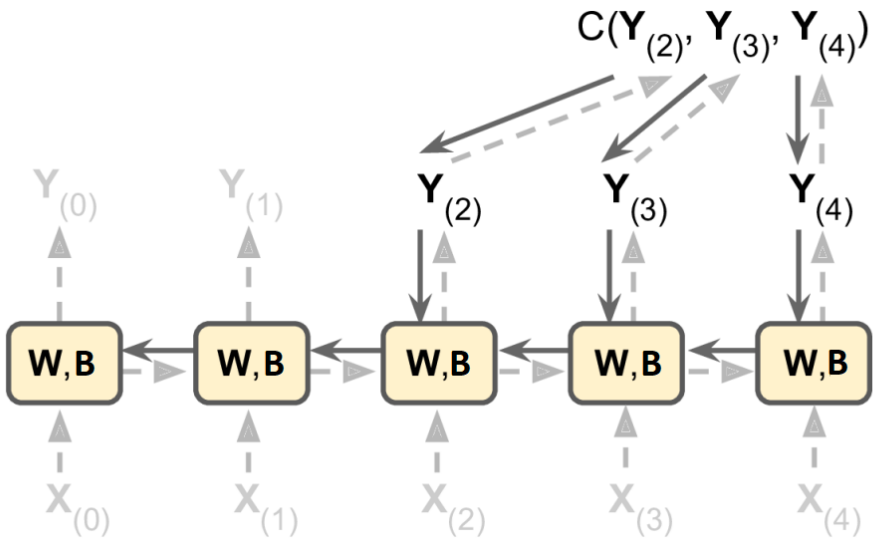

Figura 9: Retropropagação do erro através do tempo. Figura extraída de [4].

É portanto imprescindível, na maior parte dos casos, que se utilizem funções de ativação como a tangente hiperbólica (*Tanh*) ou a logística (*sigmoid*), pelo simples fato de que ambas saturam, dada uma entrada muito grande ou muito pequena (ver Figura 3). Isso contra-ataca um possível problema com o desaparecimento ou crescimento exacerbado dos gradientes.

2.2.3 LSTM

Como mencionado anteriormente, um dos maiores problemas com a RNN simples é sua memória. Devido às transformações que o dado de entrada sofre ao longo dos instantes de tempo, uma parte da informação é perdida. Eventualmente, não sobra nenhum traço das primeiras entradas.

Para abordar esse problema, diversos tipos de células com memórias a longo-prazo foram criadas. A mais popular entre elas é a LSTM [5].

Vista de fora, uma célula LSTM pode ser utilizada analogamente a um neurônio recorrente simples. A diferença é que seu estado é dividido em dois vetores diferentes: o vetor de longo prazo $\mathbf{c}$ e o vetor de curto-prazo $\mathbf{h}$. A estrutura interna de uma célula LSTM está representada na Figura 10.

A ideia principal é que a rede consiga aprender o que deve ser guardado no estado a longo prazo $\mathbf{c}_{(t)}$, e daquilo que foi guardado, o que deve ser descartado e o que deve ser considerado.

A primeira alteração que ocorre no vetor $\mathbf{c}_{(t-1)}$ é a perda de parte de sua memória, através da multiplicação elemento por elemento no que é chamado porta do esquecimento

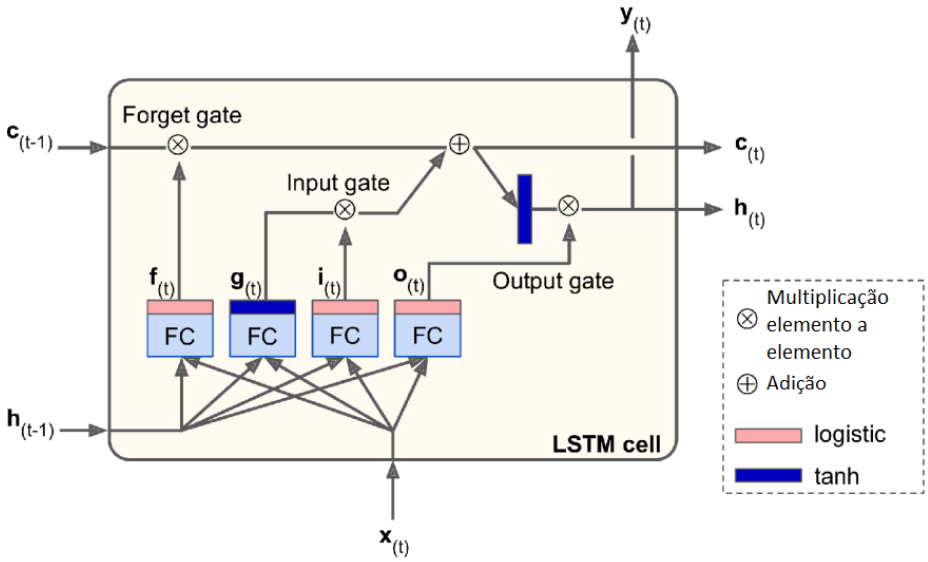

Figura 10: Estrutura interna de uma célula LSTM. Figura extraída de [4].

(do inglês, *forget gate*).

Depois, são incorporadas ao vetor novas memórias, selecionadas pela porta de entrada (do inglês, *input gate*) através da operação de adição. O resultado $\mathbf{c}_{(t)}$ é mandado então para a saída da célula. Porém, após a operação de adição, uma cópia do vetor $\mathbf{c}_{(t)}$ passa pela função tangente hiperbólica e é filtrado (novamente, com uma multiplicação termo a termo) pela porta de saída (do inglês, *output gate*). O resultado constitui o estado de curto prazo $\mathbf{h}_{(t)}$ o qual é também a saída da célula naquele instante $\mathbf{y}_{(t)}$.

Agora, olharemos para a entrada da LSTM. Tanto o vetor de entrada $\mathbf{x}_{(t)}$ quanto o vetor de estado de curto prazo do instante passado, $\mathbf{h}_{(t-1)}$, são fornecidos a quatro camadas diferentes: $\mathbf{f}_{(t)}$, $\mathbf{g}_{(t)}$, $\mathbf{i}_{(t)}$ e $\mathbf{o}_{(t)}$.

O vetor $\mathbf{g}_{(t)}$ é a saída da camada principal. Sua função é analisar a entrada $\mathbf{x}_{(t)}$ e o estado de curto-prazo prévio $\mathbf{h}_{(t-1)}$. Em uma célula simples, existe apenas esta camada, cuja saída compõe o próximo estado de curto-prazo $\mathbf{h}_{(t)}$ e a saída da própria célula naquele instante $\mathbf{y}_{(t)}$.

Em uma LSTM, por outro lado, as partes mais relevantes de $\mathbf{g}_{(t)}$ são guardadas no vetor $\mathbf{c}_{(t-1)}$, enquanto as partes menos relevantes são descartadas.

As camadas restantes são controladores, cujas saídas vão de 0 a 1 (devido à aplicação da função logística) e alimentam as portas da LSTM. Todas as saídas participam de uma multiplicação elemento a elemento, de modo que um elemento igual a 0 fecha a porta e um elemento igual a 1 abre a porta.

Deste modo, o vetor $\mathbf{f}_{(t)}$ consegue controlar quais partes do vetor $\mathbf{c}_{(t-1)}$ devem ser

apagadas. O vetor $\mathbf{i}_{(t)}$ consegue controlar quais partes de $\mathbf{g}_{(t)}$ devem ser adicionadas ao vetor $\mathbf{c}_{(t-1)}$. Finalmente, o vetor $\mathbf{o}_{(t)}$ consegue controlar quais partes de $\mathbf{c}_{(t-1)}$ devem formar tanto a saída $\mathbf{y}_{(t)}$ quanto o estado de curto-prazo $\mathbf{h}_{(t)}$.

A equação (10) mostra como computar a saída da célula, bem como os estados de curto e longo prazo, no instante t para um *mini-batch*:

$$\begin{aligned}
\mathbf{I}_{(t)} &= \sigma(\mathbf{X}_{(t)}\mathbf{W}_{xi} + \mathbf{H}_{(t-1)}\mathbf{W}_{hi} + \mathbf{B}_i) \\
\mathbf{F}_{(t)} &= \sigma(\mathbf{X}_{(t)}\mathbf{W}_{xf} + \mathbf{H}_{(t-1)}\mathbf{W}_{hf} + \mathbf{B}_f) \\
\mathbf{O}_{(t)} &= \sigma(\mathbf{X}_{(t)}\mathbf{W}_{xo} + \mathbf{H}_{(t-1)}\mathbf{W}_{ho} + \mathbf{B}_o) \\
\mathbf{G}_{(t)} &= \tanh(\mathbf{X}_{(t)}\mathbf{W}_{xg} + \mathbf{H}_{(t-1)}\mathbf{W}_{hg} + \mathbf{B}_g) \\
\mathbf{C}_{(t)} &= \mathbf{F}_{(t)} \otimes \mathbf{C}_{(t-1)}^{\top} + \mathbf{I}_{(t)} \otimes \mathbf{G}_{(t)}^{\top} \\
\mathbf{Y}_{(t)} &= \mathbf{H}_{(t)} = \mathbf{O}_{(t)} \otimes \tanh(\mathbf{C}_{(t)}^{\top})
\end{aligned} \tag{10}$$

onde σ é a função de ativação logística (*sigmoid*).

Resumidamente, uma célula LSTM é capaz de detectar uma entrada relevante (pela porta de entrada), guardá-la no vetor de estado a longo-prazo o quanto for necessário (pela porta do esquecimento) e extraí-la quando necessário (pela porta de saída). Isso traz à tona a capacidade elevada (em relação a uma célula recorrente simples) de reconhecimento e preservação de tendências presentes, no contexto deste projeto, em série temporais.

2.2.4 GRU

A célula GRU é uma versão simplificada da célula LSTM. Sua estrutura interna está representada na Figura 11.

Como pode ser notado, há agora somente um vetor de estado, **h**. Também, um único vetor $\mathbf{z}_{(t)}$ controla tanto a porta de esquecimento (*forget gate*) quanto a porta de entrada (*input gate*). Se sua saída for igual a 1, a porta de esquecimento fica aberta e a de entrada fica fechada ($1 - 1 = 0$). Se sua saída for 0, o oposto acontece. Em outras palavras, para que uma memória seja mantida, o conteúdo do local em questão deve ser primeiramente apagado.

Além disso, não existe mais uma porta de saída (*output gate*). A todo instante, a saída da célula $\mathbf{y}_{(t)}$ e seu estado atual $\mathbf{h}_{(t)}$ são resultados diretos da operação de adição. Contudo, existe agora o vetor $\mathbf{r}_{(t)}$, o qual decide qual parte do vetor de estado prévio $\mathbf{h}_{(t-1)}$ será mostrado à camada principal (de saída $\mathbf{g}_{(t)}$). É importante notar que a capacidade de memória de uma LSTM ou uma GRU é, ainda assim, limitada, de modo que o aprendizado de padrões bastante afastados no tempo pode ser difícil.

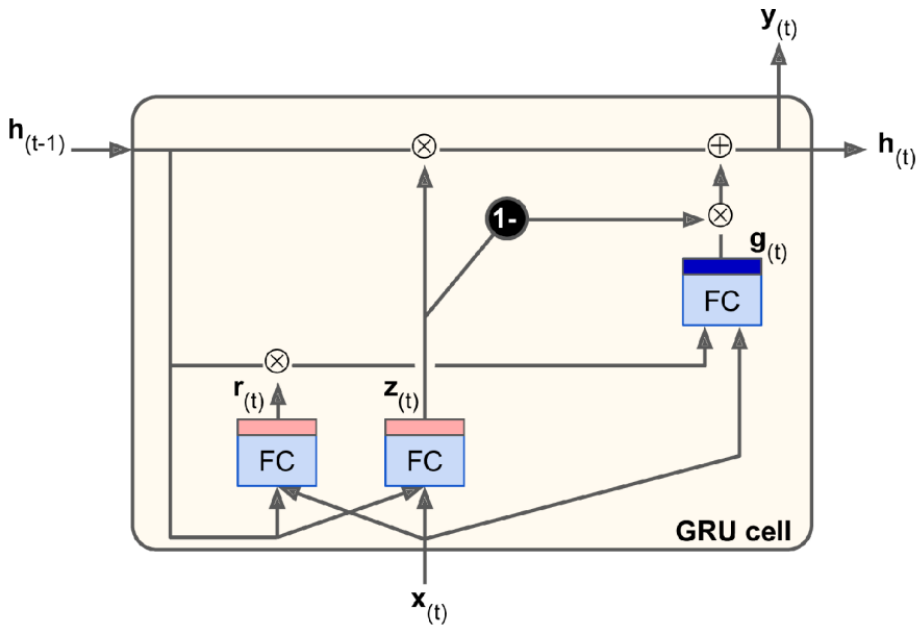

Figura 11: Estrutura interna de uma célula GRU. Figura extraída de [4].

Isto finaliza a exibição do embasamento teórico utilizado na segunda parte deste projeto. Nas próximas seções, apresentaremos características relevantes da série de Jirau, bem como a divisão de dados aplicada para o modelo *feedforward* (MLP) e para os modelos recorrentes (RNN simples, LSTM e GRU). As etapas serão explicadas de modo genérico para, depois, serem detalhadas nas exposições das implementações particulares da MLP e das RNNs, referenciando sempre a teoria discutida na Seção 2. Concomitantemente, os resultados obtidos para ambos os modelos lineares e não-lineares na etapa de teste serão relatados, analisados e, ao final, comparados entre si.

3 Metodologia e Resultados

Nesta segunda parte do projeto, foram implementados os 4 tipos de redes neurais (MLP, RNN simples, LSTM e GRU) para cada uma das séries (Água Vermelha e Jirau), assim como o modelo AR e ARMA para a série de Jirau. Estas implementações serão abordadas nas próximas seções (3.1 a 3.4) assim como a descrição da série de Jirau. Ao final, os resultados (ver Seção 3.6) expostos cobrirão o projeto como um todo, e este será concluído na Seção 4.

3.1 Descrição da Série e Divisão de Dados

A série de vazões fluviais diárias (em m^3/s) utilizada nesta segunda parte do projeto foi a da usina de Jirau, considerando-se o mesmo período de 16 anos utilizado para a série de Água Vermelha (de 1º de janeiro de 2000 a 31 de dezembro de 2015).

Novamente, os primeiros 5114 dias (1º de janeiro de 2000 a 31 de dezembro de 2013) constituíram os dados de treinamento e validação, os quais foram fornecidos ao modelo de previsão. Já os últimos 730 dias (1º de janeiro de 2014 a 31 de dezembro de 2015) foram usados na etapa de teste.

Essa série (dividida nos dois casos) pode ser observada nas Figuras 12 e 13. Ademais, algumas características da série, como média e desvio padrão, são mostradas na Tabela 1.

Tabela 1: Dados - Jirau

	Treinamento e validação	Teste
Média (m^3/s)	17772	23686
Desvio padrão (m^3/s)	11339	14917
Máxima (m^3/s)	44889	57423
Mínima (m^3/s)	2549	4547

Observando a Tabela 1, nota-se que as vazões da série de Jirau são significativamente maiores do que as de Água Vermelha, assim como seu desvio padrão e a diferença entre vazão máxima e mínima. Observando-se os gráficos nas Figuras 12 e 13, nota-se também uma curva com comportamento claramente distinto daquele apresentado pela série de Água Vermelha. Deste modo, foi possível trazer variedade às análises entre os métodos de previsão empregados neste projeto.

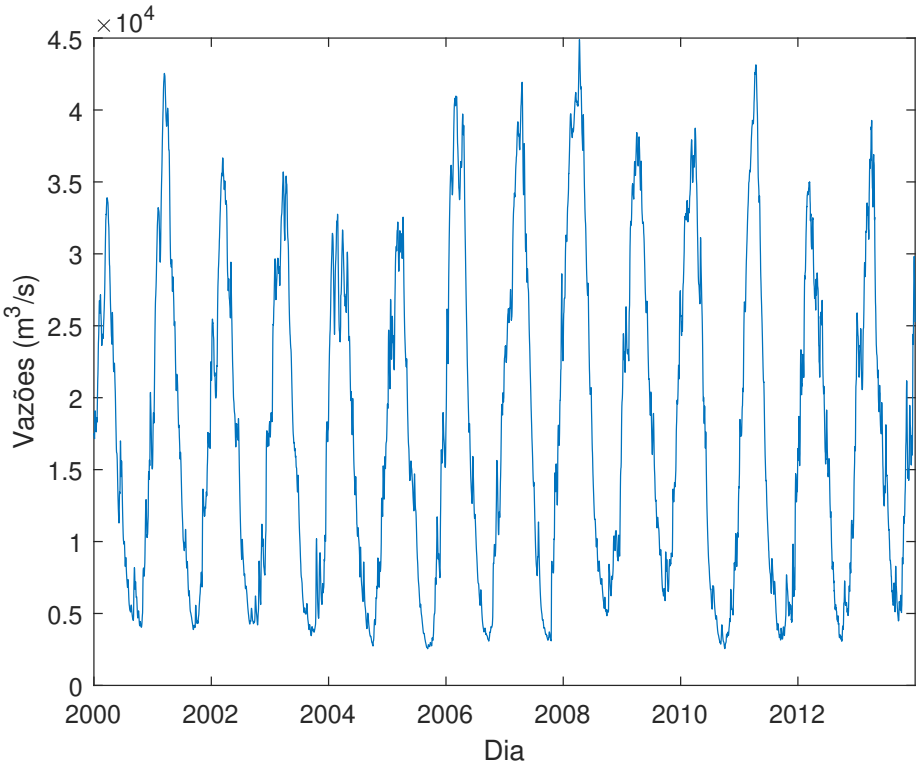

Figura 12: Série de Vazões de Jirau - Dados de treinamento.

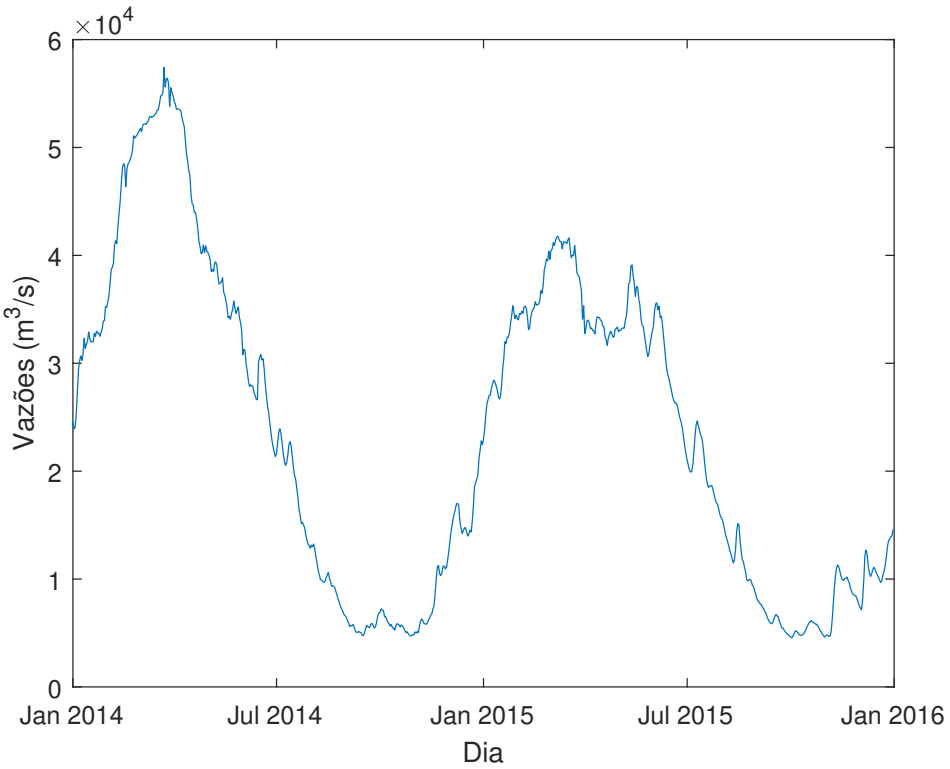

Figura 13: Série de Vazões de Jirau - Dados de teste.

3.2 Implementação

A mesma metodologia foi empregada durante o projeto e avaliação da MLP e das redes neurais recorrentes (RNN simples, LSTM e GRU), de modo a tornar verossímil a comparação de seus respectivos desempenhos. A implementação destas redes foi dividida em duas etapas: validação e teste.

Contudo, há dois pontos a serem destacados em relação à divisão de dados da MLP e das redes recorrentes. O primeiro é que, enquanto a validação cruzada (do inglês, *cross validation*) do tipo *holdout* [10] foi utilizada para as redes recorrentes (assim como nos modelos AR e ARMA), uma validação cruzada do tipo 4-*fold* foi utilizada para a MLP.

No primeiro caso, pelo fato do fator temporal ser intrínseco ao treinamento de redes recorrentes, desencourajou-se a divisão do tipo *k-fold*, pois esta quebra a sequência da série temporal. Assim, novamente os primeiros 14 anos foram divididos nos 3653 dados de treinamento (1º de janeiro de 2000 até 31 de dezembro de 2009) e nos 1461 dados de validação (1º de janeiro de 2010 até 31 de dezembro de 2013). Já na etapa de testes, foram utilizadas as medidas de vazão referentes ao período de 1º de janeiro de 2014 até 31 de dezembro de 2015 (totalizando 730 dias).

No segundo caso, a ausência do fator temporal durante o treinamento da MLP permitiu a utilização da validação-cruzada do tipo *k-fold*, na tentativa de deixar mais robusta a busca pela melhor configuração da rede durante a etapa de validação.

Na tentativa de aproximar ao máximo o tamanho do vetor de validação utilizado para a MLP do tamanho do vetor de validação utilizado por todos os outros modelos implementados, optou-se pela divisão em quatro pastas (*folds*). Esta garante um vetor de validação de tamanho 1278, comparado com o tamanho do vetor de validação utilizado na divisão *holdout* de 1461. Essa diminuição no tamanho do vetor de validação apenas implicou um aumento (de 3653 para 3836) do vetor de treinamento, sem alteração no tamanho do vetor de teste.

O segundo ponto a ser destacado trata da divisão dos dados dentro das matrizes de entrada $\mathbf{X}$ e de saída esperada $\hat{\mathbf{Y}}$ as quais devem ser introduzidas a uma rede neural durante um treinamento supervisionado.

A matriz $\mathbf{X}$ contém, em cada uma de suas linhas, uma parte (de tamanho M) diferente da série original. As partes entre si estão deslocadas de um dia, de modo que a primeira linha contenha as vazões dos dias 1 a M, a linha 2 contenha as vazões do dia 2 a $M+1$, e assim por diante. Já o vetor $\hat{\mathbf{Y}}$ contém, em suas linhas, a próxima vazão esperada (isto é, do próximo dia) em relação àquela linha, começando portanto com $M+1$, depois $M+2$, e assim por diante.

Cada linha da matriz $\mathbf{X}$ corresponde a uma instância, e M é o número de atributos (no contexto de séries temporais, o número de atrasos) contidos nas instâncias. Um conjunto de d linhas da matriz $\mathbf{X}$, portanto, define um *mini-batch*; e as d saídas esperadas deste *mini-batch* se encontram nas d linhas equivalentes do vetor $\hat{\mathbf{Y}}$.

As matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ para um mesmo número de atrasos M são as mesmas para a MLP e para as redes recorrentes. Porém, o modo como são interpretadas estas matrizes, no contexto de MLPs (Figura 4) e RNNs (Figura 9), é diferente.

No contexto de MLPs, as instâncias da matriz $\mathbf{X}$ são alimentadas, uma a uma, para todos os neurônios da primeira camada intermediária (observe a Figura 4). Já as linhas da matriz $\hat{\mathbf{Y}}$ são as saídas únicas da MLP, a qual possui um único neurônio na camada de saída.

Agora, no contexto de RNNs, as matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ definem uma rede recorrente do tipo sequência-vetor, a qual possui um único neurônio na camada de saída. O número de atrasos M equivale ao número de instantes de tempo da rede. Em outras palavras, cada um dos atributos de uma instância equivale à entrada única da rede em dado instante de tempo (observe a Figura 6), recebida por todos os neurônios da primeira camada intermediária.

Já as saídas esperadas em $\hat{\mathbf{Y}}$ equivalem ao elemento de saída da rede no último instante de tempo M considerado.

Esclarecidos os significados das divisões da série temporal nas matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ no contexto de MLPs e RNNs, cabe agora descrever as etapas de validação e teste aplicadas para estas redes neurais.

1. **Etapa de Validação**

    1.1. Escolhe-se um número de atrasos M pertencente a um intervalo I_M, e criam-se as duas matrizes $\mathbf{X}$ (uma para os dados de treinamento, e outra para os de validação) e as duas matrizes $\hat{\mathbf{Y}}$ correspondentes. No caso da MLP, o mesmo é feito, porém desta vez as matrizes $\hat{\mathbf{X}}$ e $\hat{\mathbf{Y}}$ são únicas pois os vetores de treinamento e validação foram concatenados (em um de tamanho 3653) para esta etapa. O motivo por trás disto será explicado na Seção 3.3.

    1.2. Criam-se redes neurais (do tipo em questão) a partir da biblioteca com uma camada intermediária de $N \subset I_N$ neurônios (I_N variando com o tipo da rede) e uma camada de saída contendo um único neurônio. Treinam-se as redes construídas por um número de épocas, sendo seus desempenhos o menor EQM cometido para os dados de validação. Como a MLP possui 4 conjuntos

distintos de validação (cada um correspondendo a 1/4 do vetor concatenado), o desempenho das redes criadas é dado pela média dos 4 (um para cada conjunto) EQMs finais de validação. Define-se, então, o número N de neurônios que obteve o melhor desempenho para aquele número de atrasos.

1.3. O processo é repetido para outros valores de M pertencentes ao intervalo I_M. Ao final, consideramos os três melhores números de atrasos M e seus números de neurônios correspondentes;

2. **Etapa de Teste**

2.1. As melhores configurações de cada rede (indicadas por uma combinação de número de atrasos e número de neurônios) são, então, utilizadas no retreinamento, feito desta vez com validação cruzada do tipo *holdout*. Não houve alteração na divisão de dados, isto é, os dados de validação e os de treinamento correspondem aos mesmos 1461 e 3653 dias, respectivamente. No caso da MLP, o retreinamento é feito de duas maneiras: a primeira utilizando-se a validação cruzada do tipo *holdout* como descrito, e a segunda utilizando os vetores de treinamento e validação concatenados, sem validação.

2.2. Os modelos retreinados são usados para prever todas as vazões presentes nos dados de teste (de 1º de janeiro de 2014 a 31 de dezembro de 2015). Baseado nas saídas geradas, calcula-se o EQM cometido por cada conjunto.

2.3. O retreinamento é feito um total de 5 vezes para cada conjunto. A menor média dos EQMs para os dados de teste produzida entre os conjuntos é a medida do desempenho daquela rede na previsão da série em questão. A partir dela é feita a comparação de sua eficiência perante os outros modelos.

A descrição específica da implementação de cada modelo será detalhada nas Seções 3.3 e 3.4.

3.2.1 Motivação para a escolha dos atrasos M e K

Pelo mesmo motivo apresentado no relatório parcial, o número de atrasos M foi definido, a princípio, através da função de autocorrelação parcial aplicada sobre os dados de treinamento e validação concatenados da série de Jirau. Seu gráfico pode ser visto na Figura 14.

Analisando a Figura 14, o maior intervalo que considera uma maioria de pontos na região de confiança e de autocorrelação significativa é o que começa em 60 e termina em 110.

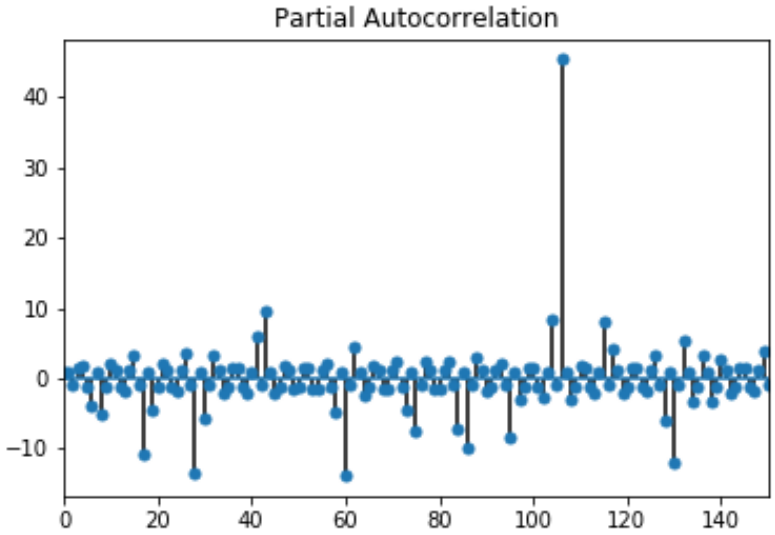

Figura 14: Autocorrelação parcial dos dados de treinamento e validação.

Contudo, após realizadas as etapas de validação para todos os modelos lineares e não-lineares utilizando-se valores de M iguais a 60, 110 e 150, foram obtidos valores do menor EQM de validação maiores do que quando M tinha valor muito menor. Deste modo, optou-se, como no relatório parcial, pelo intervalo $I_M = [1, 30]$.

Já o intervalo de valores possíveis para K foi definido novamente a partir da função de autocorrelação para os dados de treinamento e validação referente à série de Jirau, vista na Figura 15.

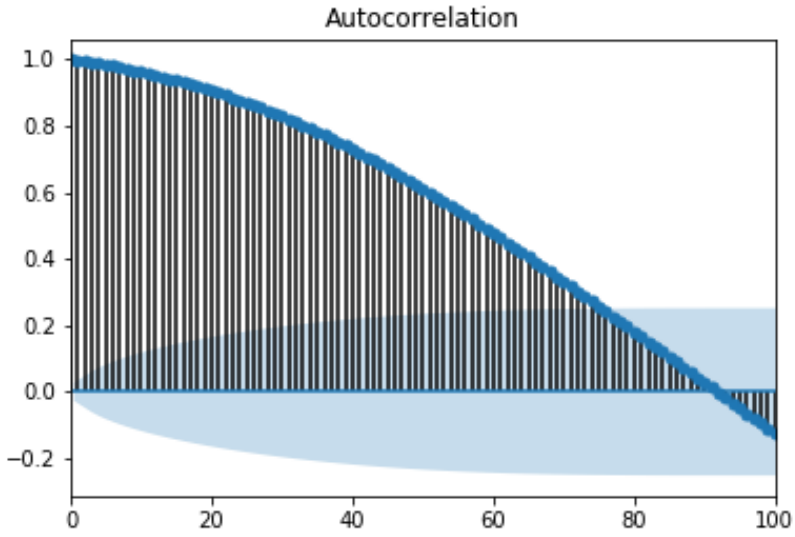

Figura 15: Autocorrelação dos dados de treinamento e validação.

Observa-se na Figura 15 que, a partir do atraso igual a 70, deixa-se o intervalo de

confiança (fora da região azul) e os valores da autocorrelação se tornam muito baixos.

A princípio, o intervalo de consideração iria de $K = 1$ até $K = 70$. Porém, após realizados testes com o modelo ARMA, conclui-se que, para diversos valores de M pertencentes a I_M, um $K \geq 40$ sempre levava a saídas divergentes. Por isso, optamos pelo mesmo conjunto de atrasos do relatório parcial, $C_K = \{1, 5, 10, 20, 40\}$.

3.3 Implementação da MLP

Para a etapa de validação (ver Seção 3.2), deseja-se encontrar o valor de atraso M dentro de I_M (ver Seção 3.2.1) que leve ao menor EQM para os dados de validação.

Foi necessário apenas um laço externo que percorresse o intervalo I_M. Para dado número de colunas M, são construídas as matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ a partir dos vetores de validação e treinamento concatenados (3653 dados).

A biblioteca Keras escrita em Python possui a função GridSearchCV, a qual realiza por completo a etapa de validação descrita na Seção 3.2. Como sugere seu nome, ela aplica, por *default*, a validação cruzada do tipo *k-fold*. Assim, basta entregar o conjunto completo (no caso, o concatenado) das matrizes de entradas e saídas esperadas, pois a própria função divide-as nos 4 pares de treinamento e validação desejados.

Para que as diferentes redes com diferentes números N de neurônios sejam criadas, devem-se, antes, definir os parâmetros comuns (e invariáveis) entre elas. São tais parâmetros: a função de ativação, o otimizador, o tamanho do *batch* (em inglês, *batch size*), a inicialização de dados, a taxa de aprendizado (em inglês, *learning rate*) e a presença ou ausência do *batch normalization*.

Considerando todas as 4 redes neurais e os valores de $M \subset I_M$ no contexto da série de Água Vermelha, foi feita uma busca entre as seguintes possibilidades:

1. Função de ativação: ReLU, Tanh, Sigmoid, Leaky ReLU, SELU;
2. Otimizador: RMSProp, Adam, Nadam;
3. *Batch size*: 8, 16, 32, 64, 128;
4. Inicialização de dados: He Normal, Glorot Normal, Glorot Uniform, Lecun Normal;
5. Taxa de aprendizado (η): valores com ordens de 10^{-2} a 10^{-5}.

Apenas algumas combinações dos parâmetros listados foram testadas. O melhor conjunto encontrado, o qual incorporou o *batch normalization*, pode ser visto na Tabela 2.

Tabela 2: Parâmetros da MLP

Função de ativação	η	*Batch size*	Otimizador	Inicialização
Selu	0,003	32	Nadam	Lecun Normal

Definidos os parâmetros fixos das redes, a rotina GridSearchCV as treina por 150 épocas variando o número de neurônios dentro do conjunto $I_N^{\text{MLP}} = \{5, 10, 20, 30, 50, 80\}$.

Os resultados da etapa de validação para todos números de atrasos M equivalem ao menor EQM médio final para os dados de validação e seu número N de neurônios correspondente. Eles se encontram nas Tabelas 3 (Água Vermelha) e 4 (Jirau).

Tabela 3: Resultados da Etapa de Validação - MLP - Água Vermelha

M	N	EQM $(m^3/s)^2$
1	5	18297 ± 2979
2	30	8050 ± 3076
3	5	8072 ± 3054
4	20	7863 ± 3034
5	30	7891 ± 2772
6	10	7628 ± 3084
7	20	7986 ± 4015
8	10	7479 ± 3204
9	20	7676 ± 2895
10	5	7504 ± 3106
11	5	7580 ± 3067
12	5	7521 ± 2983
13	30	7510 ± 2935
14	30	7477 ± 3146
15	50	7557 ± 3040
16	10	7786 ± 2736
17	80	7713 ± 3216
18	5	8129 ± 4251
19	30	7844 ± 2950
20	10	7466 ± 2888
21	5	8358 ± 4573
22	20	7918 ± 2710
23	5	7707 ± 3072
24	10	8118 ± 2745
25	30	7716 ± 3015
26	30	8418 ± 3507
27	20	8355 ± 2885
28	10	8009 ± 2209
29	10	8764 ± 3011
30	10	8341 ± 3302

Tabela 4: Resultados da Etapa de Validação - MLP - Jirau

M	N	EQM $(m^3/s)^2$
1	5	262048 ± 35865
2	5	105087 ± 36448
3	80	101630 ± 50732
4	5	98979 ± 41550
5	5	98979 ± 41550
6	5	97808 ± 37452
7	20	104796 ± 46494
8	80	110279 ± 42140
9	10	104156 ± 44553
10	20	103430 ± 35122
11	5	108561 ± 41719
12	30	110287 ± 36668
13	20	104084 ± 36923
14	20	112702 ± 31688
15	5	118326 ± 45031
16	10	142881 ± 73898
17	80	139666 ± 51900
18	10	120468 ± 36568
19	30	147058 ± 40942
20	20	128005 ± 38448
21	5	128557 ± 33387
22	30	122240 ± 29810
23	10	136808 ± 46225
24	50	153389 ± 41336
25	30	138478 ± 44744
26	30	136619 ± 58360
27	10	149237 ± 77826
28	20	168909 ± 57593
29	30	140644 ± 28467
30	80	139826 ± 25874

Como os valores dos EQMs apresentam grande variação (isto é, valores de desvio padrão significativos), foram escolhidos para o retreinamento os 3 melhores conjuntos $\{M, N\}$ para cada série.

Os três melhores modelos foram retreinados cinco vezes, considerando tanto uma abordagem com validação cruzada do tipo *holdout*, quanto uma abordagem sem validação (na qual todos os dados de treinamento e validação servem para treinamento). Os EQMs médios obtidos por estes modelos, para ambas as séries de vazões consideradas no trabalho, estão apresentados nas Tabelas 5, 6, 7 e 8. O melhor resultado para cada série está destacado em negrito.

Tabela 5: Resultados da Etapa de Teste (vetor concatenado) - MLP - Água Vermelha

M	N	EQM 1	EQM 2	EQM 3	EQM 4	EQM 5	EQM médio
8	10	1284	1144	1198	1120	1078	$\mathbf{1165 \pm 79}$
14	30	1184	2573	1255	1859	1165	1607 ± 611
20	10	1126	1176	1140	1226	1287	1191 ± 66

Tabela 6: Resultados da Etapa de Teste (*holdout*) - MLP - Água Vermelha

M	N	EQM 1	EQM 2	EQM 3	EQM 4	EQM 5	EQM médio
8	10	1465	1109	1164	1267	1590	1319 ± 203
14	30	1216	1222	1161	1105	1162	$\mathbf{1173 \pm 48}$
20	10	1141	1474	1306	1122	1245	1257 ± 143

Tabela 7: Resultados da Etapa de Teste (vetor concatenado) - MLP - Jirau

M	N	EQM 1	EQM 2	EQM 3	EQM 4	EQM 5	EQM médio
4	5	468495	481654	475733	485869	479102	$\mathbf{478170 \pm 6554}$
5	5	483917	455231	571951	479152	454561	488962 ± 48296
6	5	441881	477523	573238	438077	579538	502051 ± 37961

Tabela 8: Resultados da Etapa de Teste (*holdout*) - MLP - Jirau

M	N	EQM 1	EQM 2	EQM 3	EQM 4	EQM 5	EQM médio
4	5	582427	861541	665680	568981	607693	657264 ± 120053
5	5	600882	667731	563278	563221	559239	590858 ± 46187
6	5	514205	453975	547496	466437	508845	$\mathbf{498191 \pm 37961}$

Destacamos os menores EQMs para o conjunto de teste de ambas as séries na Tabela 9.

Tabela 9: Resultados Finais - MLP

Série	EQM $(m^3/s)^2$
Água Vermelha	1165 ± 79
Jirau	478170 ± 6554

O desempenho alcançado pela MLP pode ser considerado satisfatório, embora esteja um pouco acima do desempenho oferecido pelos modelos AR e ARMA.

3.4 Implementação das RNNs

O mesmo procedimento aqui descrito foi aplicado para as três redes recorrentes (RNN simples, LSTM e GRU), uma vez que, baseando-se na biblioteca Keras, basta substituir o tipo de célula desejado para o treinamento de uma rede.

Novamente, para a etapa de validação (ver Seção 3.2), deseja-se encontrar o valor de atraso M dentro de I_M (ver Seção 3.2.1) que leve ao menor EQM para os dados de validação.

Foi necessário apenas um laço externo que percorresse o intervalo I_M. Para dado número de colunas M, são construídas as matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ a partir dos dados de treinamento e validação separados, pois a validação desta vez é do tipo *holdout*.

Os parâmetros fixos utilizados para as redes recorrentes podem ser vistos na Tabela 10.

Tabela 10: Parâmetros das Redes Recorrentes

Função de ativação	η	*Batch size*	Otimizador	Inicialização
Tanh	0,01	32	Nadam	Glorot Uniform

Observe que, neste caso, não é feito o *batch normalization*, a taxa de aprendizado é de $0,01$ e a função de ativação utilizada é a tangente hiperbólica, assim como a inicialização é do tipo Glorot Uniform (estas duas últimas opções fazem parte do padrão (*default*) para redes recorrentes).

Como a função tangente hiperbólica apresenta saturação para entradas de módulos elevados (ver Figura 3), as matrizes $\mathbf{X}$ e $\hat{\mathbf{Y}}$ foram escaladas para valores em torno de 0.

Ainda assim, as redes recorrentes apresentaram resultados insatisfatórios, com EQMs para os dados de teste muito acima daqueles obtidos pela MLP. Decidiu-se, portanto, realizar a diferenciação dos elementos da série antes que fossem definidas as matrizes $\mathbf{X}$

e $\hat{\mathbf{Y}}$. Esta técnica consiste em subtrair cada valor da série pelo seu elemento anterior, na tentativa de remover tendências ali presentes.

Após a diferenciação e, em seguida, o escalamento das vazões, as quatro matrizes são construídas e entregues à função GridSearchCV, duas como dados de treinamento e as outras como dados de validação.

Os conjuntos contendo os números N de neurônios considerados são:

1. $I_N^{\text{RNNsimples}} = I_N^{\text{GRU}} = \{5, 10, 15, 20, 30, 50, 75, 100\}$;
2. $I_N^{\text{LSTM}} = \{20, 30, 50, 75, 100, 150, 200, 300, 500\}$.

Encontrou-se o melhor número N de neurônios para cada atraso M, concluindo a etapa de validação. Seus resultados se encontram nas Tabelas 11 (Água Vermelha - RNN simples), 12 (Água Vermelha - LSTM), 13 (Água Vermelha - GRU), 14 (Jirau - RNN simples), 15 (Água Vermelha - LSTM) e 16 (Água Vermelha - GRU).

Tabela 11: Resultados da Etapa de Validação - RNN simples - Água Vermelha

M	N	EQM $(m^3/s)^2$
1	10	0,006420
2	15	0,006067
3	20	0,005918
4	15	0,005886
5	20	0,005578
6	5	0,006080
7	10	0,006469
8	20	0,006153
9	10	0,006353
10	10	0,006420
11	5	0,006643
12	15	0,006016
13	15	0,006603
14	15	0,005595
15	10	0,006723
16	15	0,006205
17	5	0,006857
18	15	0,006648
19	15	0,006755
20	5	0,006767
21	10	0,006713
22	10	0,006399
23	5	0,006476
24	5	0,006515
25	5	0,007206
26	10	0,006654
27	5	0,006887
28	5	0,006978
29	5	0,007391
30	5	0,007185

Tabela 12: Resultados da Etapa de Validação - LSTM - Água Vermelha

M	N	EQM $(m^3/s)^2$
1	300	0,006200
2	75	0,005332
3	30	0,004737
4	150	0,003875
5	300	0,003198
6	200	0,002793
7	150	0,002649
8	150	0,001862
9	100	0,002541
10	200	0,002261
11	300	0,002299
12	100	0,002684
13	150	0,002273
14	150	0,002380
15	150	0,002246
16	150	0,002669
17	300	0,002970
18	100	0,002796
19	100	0,003549
20	30	0,003471
21	150	0,003098
22	100	0,002843
23	50	0,004476
24	75	0,003643
25	50	0,003351
26	30	0,003840
27	50	0,004041
28	200	0,004046
29	100	0,004620
30	50	0,004739

Tabela 13: Resultados da Etapa de Validação - GRU - Água Vermelha

M	N	EQM $(m^3/s)^2$
1	30	0,006220
2	75	0,005303
3	30	0,004949
4	30	0,004330
5	30	0,003709
6	50	0,003815
7	100	0,003653
8	75	0,003748
9	75	0,003463
10	100	0,003320
11	30	0,004382
12	50	0,003502
13	75	0,004063
14	30	0,004075
15	100	0,004884
16	100	0,003638
17	100	0,004820
18	10	0,004939
19	10	0,005034
20	100	0,005134
21	75	0,004384
22	20	0,005026
23	30	0,004401
24	20	0,005451
25	100	0,004625
26	100	0,005643
27	50	0,006018
28	50	0,005872
29	20	0,005122
30	30	0,004763

Tabela 14: Resultados da Etapa de Validação - RNN simples - Jirau

M	N	EQM $(m^3/s)^2$
1	5	0,010417
2	50	0,008233
3	10	0,008046
4	5	0,007830
5	10	0,008000
6	10	0,007829
7	30	0,007829
8	30	0,007866
9	30	0,008103
10	20	0,008173
11	5	0,008027
12	10	0,008200
13	15	0,008193
14	5	0,008346
15	15	0,008156
16	10	0,008258
17	15	0,008435
18	10	0,007912
19	10	0,008230
20	5	0,008657
21	10	0,010516
22	10	0,008302
23	5	0,009195
24	15	0,008205
25	5	0,008446
26	10	0,008745
27	15	0,008036
28	5	0,008256
29	5	0,018116
30	10	0,007995

Tabela 15: Resultados da Etapa de Validação - LSTM - Jirau

M	N	EQM $(m^3/s)^2$
1	150	0,010399
2	200	0,007972
3	75	0,007524
4	200	0,006666
5	100	0,005998
6	75	0,004482
7	100	0,004518
8	150	0,003824
9	100	0,003819
10	75	0,003533
11	200	0,003504
12	300	0,003329
13	150	0,003032
14	150	0,003309
15	200	0,003280
16	150	0,003497
17	300	0,002710
18	150	0,002979
19	150	0,002234
20	150	0,003959
21	50	0,004506
22	100	0,002732
23	75	0,004781
24	75	0,005529
25	75	0,005042
26	100	0,005388
27	50	0,005567
28	100	0,005339
29	100	0,006223
30	75	0,004432

Tabela 16: Resultados da Etapa de Validação - GRU - Jirau

M	N	EQM $(m^3/s)^2$
1	10	0,010412
2	20	0,008050
3	50	0,007597
4	100	0,006877
5	50	0,006410
6	50	0,006080
7	100	0,005872
8	30	0,005897
9	100	0,005024
10	75	0,005309
11	30	0,005625
12	75	0,004683
13	50	0,005648
14	50	0,005557
15	30	0,005937
16	30	0,005946
17	100	0,005236
18	30	0,005810
19	75	0,005936
20	30	0,005913
21	50	0,006430
22	20	0,006283
23	20	0,006712
24	100	0,006804
25	30	0,006075
26	75	0,006629
27	20	0,006771
28	20	0,007604
29	20	0,007240
30	20	0,007012

Novamente, foram escolhidas para o retreinamento as 3 melhores combinações $\{M, N\}$ de cada rede recorrente para cada série. Os modelos foram retreinados 5 vezes (desta vez apenas com validação do tipo *holdout*) e produziram EQMs médios para os dados de teste apresentados nas Tabelas 17 (Água Vermelha - RNN simples), 18 (Água Vermelha - LSTM), 19 (Água Vermelha - GRU), 20 (Jirau - RNN simples), 21 (Jirau - LSTM) e 22 (Jirau - GRU). O melhor resultado para cada série e modelo está destacado em negrito.

Tabela 17: Resultados da Etapa de Teste - RNN simples - Água Vermelha

M	N	EQM 1	EQM 2	EQM 3	EQM 4	EQM 5	EQM médio
4	15	982	1043	1149	1091	1028	1059 ± 64
5	20	958	992	1206	1119	963	$\mathbf{1048 \pm 110}$
14	15	2130	1232	1178	1795	1505	1568 ± 399

Tabela 18: Resultados da Etapa de Teste - LSTM - Água Vermelha

M	N	EQM 1	EQM 2	EQM 3	EQM 4	EQM 5	EQM médio
8	150	1175	1279	1350	1441	1362	$\mathbf{1322 \pm 100}$
10	200	1400	1442	1713	1496	1569	1524 ± 123
15	150	6168	3682	11602	3681	1134	5254 ± 3972

Tabela 19: Resultados da Etapa de Teste - GRU - Água Vermelha

M	N	EQM 1	EQM 2	EQM 3	EQM 4	EQM 5	EQM médio
10	100	1016	1118	965	1130	1212	$\mathbf{1088 \pm 98}$
9	75	1014	1164	1130	1441	1354	1220 ± 173
12	50	1160	1011	1274	1212	1051	1141 ± 109

Tabela 20: Resultados da Etapa de Teste - RNN simples - Jirau

M	N	EQM 1	EQM 2	EQM 3	EQM 4	EQM 5	EQM médio
4	5	418153	417588	415601	425028	394600	$\mathbf{414914 \pm 11516}$
6	10	422913	417947	398628	422622	415919	415606 ± 9956
7	30	414306	474045	448726	420339	407247	432932 ± 27861

Tabela 21: Resultados da Etapa de Teste - LSTM - Jirau

M	N	EQM 1	EQM 2	EQM 3	EQM 4	EQM 5	EQM médio
17	300	481797	648852	523012	498273	401681	510723 ± 89600
19	150	515892	478438	569726	543675	557248	532996 ± 36471
22	100	450369	466372	477872	423335	462689	$\mathbf{456127 \pm 20792}$

Tabela 22: Resultados da Etapa de Teste - GRU - Jirau

M	N	EQM 1	EQM 2	EQM 3	EQM 4	EQM 5	EQM médio
9	100	529812	475181	489579	557155	459901	$\mathbf{502325 \pm 40190}$
12	75	462723	490505	450089	628442	797920	565936 ± 147891
17	100	447768	426985	485307	741512	545553	529425 ± 126836

Destacamos os melhores conjuntos $\{M, N\}$ e seus respectivos EQMs para os dados de teste para os 6 casos na Tabela 23.

Tabela 23: Resultados Finais - Redes Recorrentes

Água Vermelha	
Célula	EQM $(m^3/s)^2$
RNN simples	1048 ± 110
LSTM	1322 ± 100
GRU	1088 ± 98
Jirau	
Célula	EQM $(m^3/s)^2$
RNN simples	414914 ± 11516
LSTM	456127 ± 20792
GRU	502325 ± 40190

É possível constatar que o modelos recorrentes apresentaram um desempenho superior, no geral, àquele da MLP. Uma comparação mais detalhada entre todos os modelos será feita na Seção 3.6.

3.5 Resultados - Modelos Lineares

A fim de complementar a análise comparativa nesta segunda parte do projeto, repetimos o procedimento de construção e teste dos modelos AR e ARMA, detalhado no relatório

parcial, considerando, agora, a série de Jirau.

O desenvolvimento dos métodos lineares e todos os parâmetros relevantes (à mostra na Tabela 3 do relatório parcial) são exatamente os mesmos que foram empregados no primeiro relatório (ver Seção 3.2.1).

Começando com o modelo AR, apresentamos o resultado da etapa de validação na forma de um gráfico do EQM de validação em função do número de atrasos, o qual pode ser visto na Figura 16.

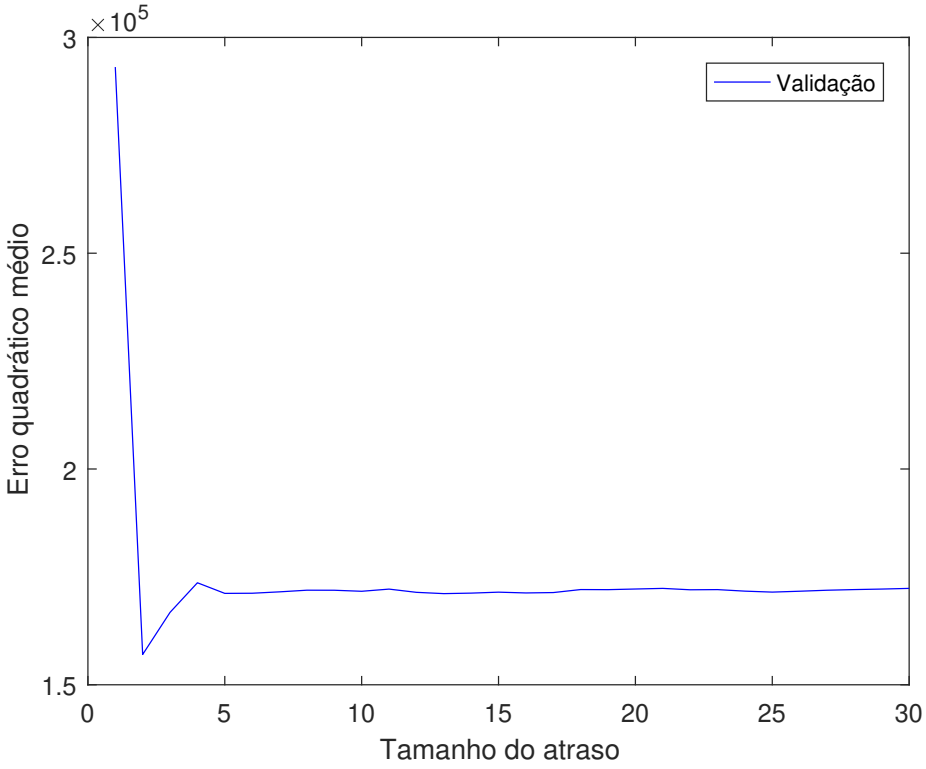

Figura 16: Erro quadrático médio por tamanho do atraso - AR.

O atraso com melhor desempenho, escolhido para o retreinamento, foi de $M = 2$. A Tabela 24 contém o EQM e o EAM para os dados de teste do modelo.

Tabela 24: Resultados - Modelo AR.

Erro Quadrático Médio $(m^3/s)^2$	393130
Erro Absoluto Médio (m^3/s)	404

O gráfico que sobrepõe as vazões originais de Jirau com aquelas previstas na etapa de teste pelo modelo AR é mostrado na Figura 17. Com o objetivo de melhorar a visualização, o gráfico de previsões em um período agora de 1 mês é mostrado na Figura 18.

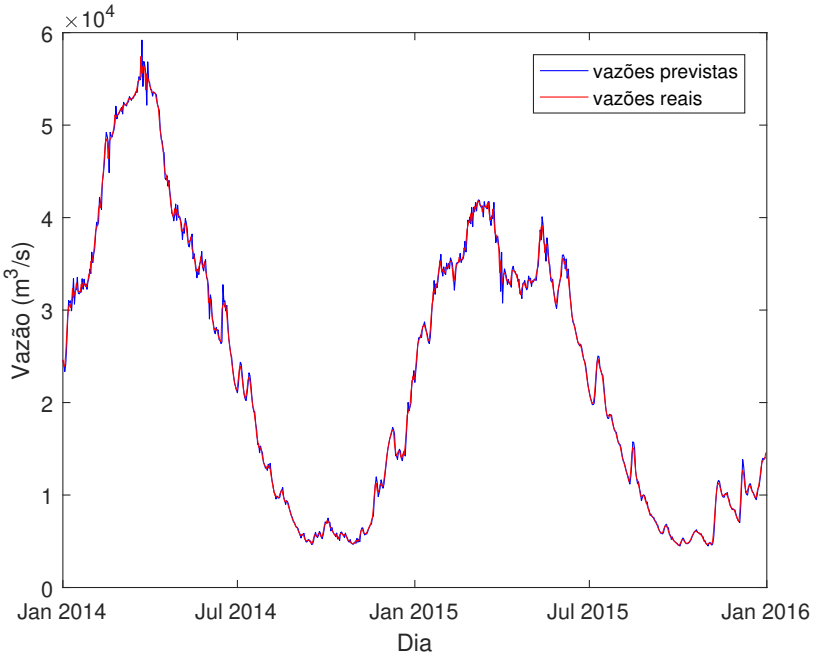

Figura 17: Vazões previstas e vazões reais - AR.

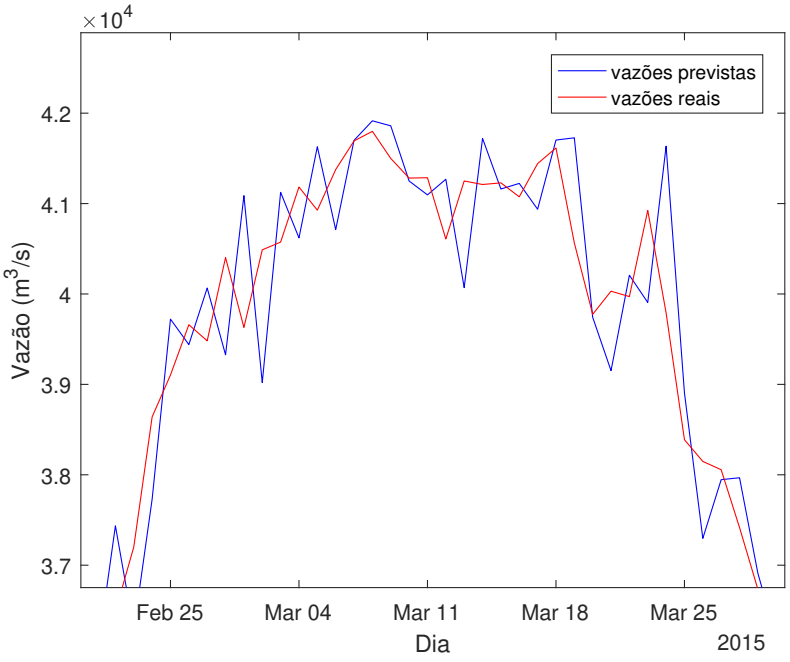

Figura 18: Vazões previstas e vazões reais com *zoom* - AR.

Para o modelo ARMA, a Tabela 25 apresenta os valores médios dos EQMs obtidos na etapa de validação para as combinações de M e K consideradas. A princípio, nota-se uma grande diferença entre os valores (indo da ordem de 10^5 até 10^7). Por essa razão, decidimos desconsiderar os valores de ordem acima de 10^6 (por serem divergentes) para plotar o histograma da Figura 19.

Tabela 25: Resultados da Validação - Modelo ARMA

M	K	EQM $(m^3/s)^2$
1	1	182980 ± 40
1	5	151300 ± 2190
1	10	151210 ± 2130
1	20	190700 ± 26930
1	40	13886000 ± 30058000
5	1	149530 ± 950
5	5	155810 ± 8120
5	10	195330 ± 25600
5	20	368050 ± 63360
5	40	2503700 ± 1920200
15	1	204300 ± 10990
15	5	203350 ± 45280
15	10	217260 ± 44420
15	20	411530 ± 67710
15	40	10308000 ± 9836000
30	1	316600 ± 18470
30	5	289440 ± 75580
30	10	358380 ± 122430
30	20	538000 ± 143260
30	40	3644800 ± 2434900

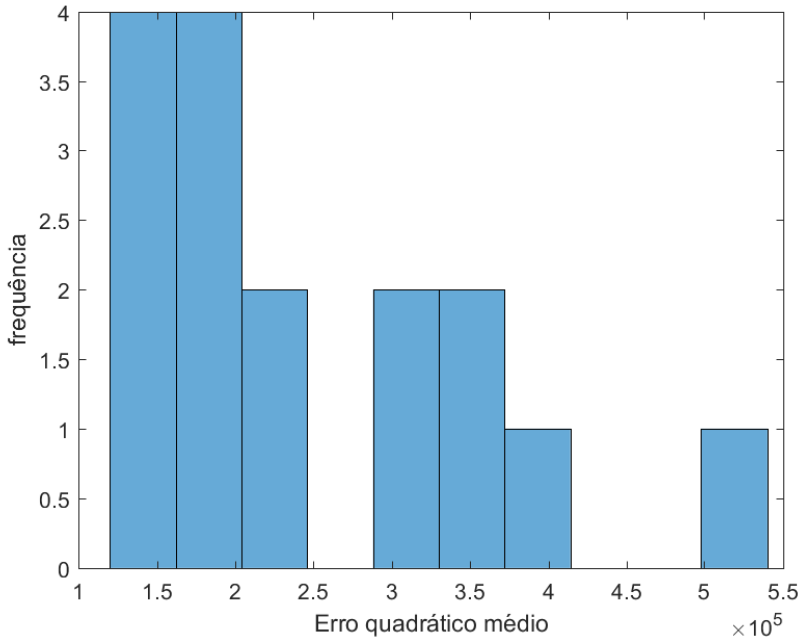

Figura 19: Histograma do erro quadrático médio - ARMA.

A combinação de atrasos selecionada para o retreinamento foi de $M = 5$ e $K = 1$. A Tabela 26 contém o EQM e o EAM para os dados de teste do modelo.

Tabela 26: Resultados do Teste - Modelo ARMA

Erro Quadrático Médio $(m^3/s)^2$	614060
Erro Absoluto Médio (m^3/s)	489

Por sua vez, o gráfico que sobrepõe as vazões originais de Jirau com aquelas previstas na etapa de teste pelo modelo ARMA é mostrado na Figura 20. Com o objetivo de melhorar a visualização, o gráfico de previsões em um período agora de 1 mês é mostrado na Figura 21.

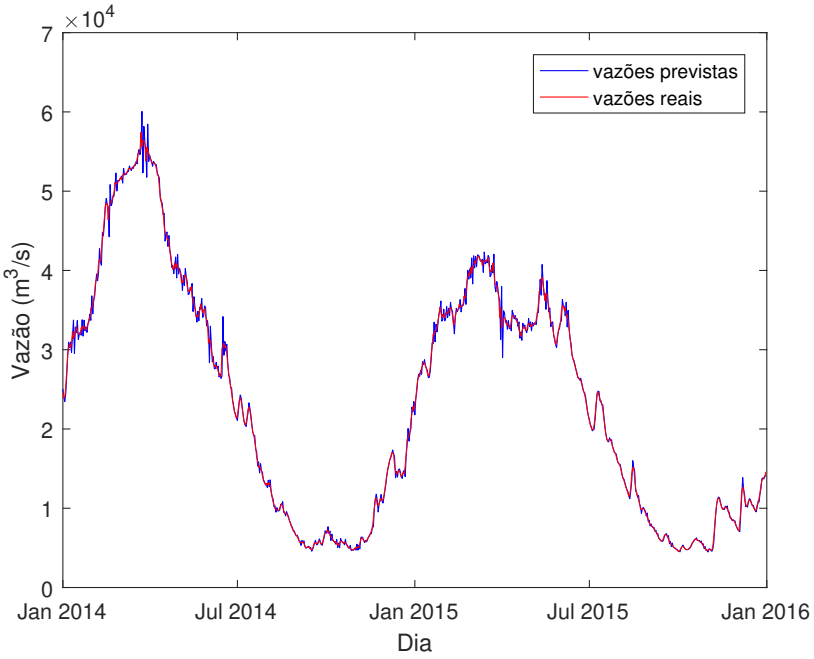

Figura 20: Vazões previstas e vazões reais - ARMA.

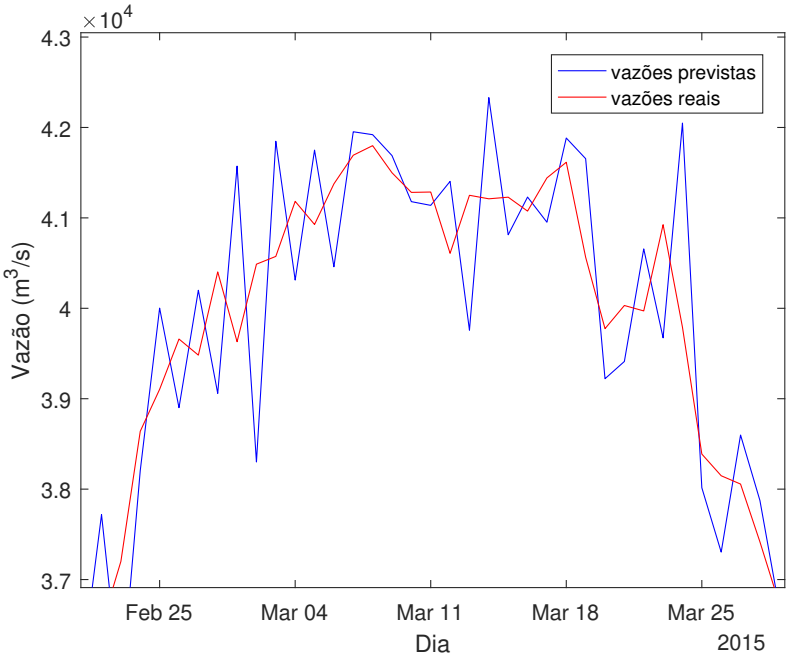

Figura 21: Vazões previstas e vazões reais com *zoom* - ARMA.

3.6 Comparação entre modelos

Os EQMs para os dados de teste de cada modelo são colocados na Tabela 27 em ordem crescente.

O primeiro ponto a ser destacado é que o desempenho das células recorrentes foi

Tabela 27: Resultados Finais

Água Vermelha	
Modelo	EQM $(m^3/s)^2$
RNN simples	1048 ± 110
ARMA	1056,3
GRU	1088 ± 98
AR	1088,6
MLP	1165 ± 79
LSTM	1322 ± 100
Jirau	
Modelo	EQM $(m^3/s)^2$
AR	393130
RNN simples	414914 ± 11516
LSTM	456127 ± 20792
MLP	478170 ± 6554
GRU	502325 ± 40190
ARMA	614060

superior ao desempenho da MLP. Consideremos uma MLP com uma única camada de N_n neurônios. Esta rede contém, para instâncias de tamanho n, $n \times N_n$ pesos a serem ajustados. Já uma rede recorrente de camada única, e mesmo número de neurônios, necessita apenas ajustar N_n parâmetros (independentemente do tamanho da instância), pois estes são compartilhados entre os n instantes de tempo. O número reduzido de pesos da rede recorrente simplifica seu aprendizado.

Quanto ao acesso das redes à instância, de fato, a MLP a observa por completo, enquanto a rede recorrente acessa somente um de seus elementos em cada instante de tempo. A informação introduzida por cada um desses elementos é passada para os instantes de tempo subsequentes apenas pelo estado da célula, dificultando a assimilação de padrões longos. Contudo, como as instâncias apresentam um tamanho $n \in I_M$ reduzido, a célula de memória consegue aprender com êxito os padrões fundamentais presentes na instância. Isto minimiza a possível vantagem possuída pela MLP, de acesso à instância completa, e configura outro motivo pelo qual o desempenho da RNN simples foi formidável.

Por outro lado, tanto a LSTM quanto a GRU não ofereceram melhorias em relação à RNN simples. Aquelas possuem a habilidade de manter uma informação dentro da rede

por períodos de tempo mais longos. Porém, os períodos de tempo considerados (tamanho das instâncias) eram curtos, e, portanto, o gasto de recursos com a arquitetura complexa da memória a longo prazo provou ser infrutífero para a otimização destas células. Assim, tanto a LSTM como a GRU foram superadas pela RNN simples.

Os modelos lineares, por sua vez, se saíram muito bem. O estudo feito em [11] mostra que modelos estatísticos (como o AR e o ARMA) ~~com baixos custos computacionais superam, comumente,~~ métodos de aprendizado de máquina (ML, em inglês *Machine Learning*), cujas sofisticação e demanda computacional são maiores. Isso mostra que um alto desempenho de extrapolação é acompanhado de grande eficiência. Em outras palavras, uma capacidade superior de predição pode ser atingida a partir de uma menor complexidade computacional (CC, em inglês *Computational Complexity*). O inverso também é verdadeiro, já que abordagens mais complexas de extrapolação por métodos de ML exibem resultados piores, indicando que robustez teórica nem sempre é favorável.

O estudo [11] também mostra que modelos que melhor ajustam os parâmetros da rede durante o treinamento não necessariamente apresentam maior capacidade de extrapolação. O exemplo dado pelo artigo é a LSTM, a qual "comparada com redes neurais mais simples, como a RNN e a MLP, demonstrou melhor *fitting* porém pior precisão de predição [para os dados de teste]". Comparando-se, neste projeto, o desempenho de validação da LSTM (ver Tabelas 12 e 15) ao desempenho de validação da RNN simples (ver Tabelas 11 e 14), nota-se esse mesmo comportamento: os EQMs de validação da LSTM são, no geral, menores, porém a capacidade de extrapolação da RNN simples é superior no contexto de ambas as séries.

Em suma, as características das séries de Água Vermelha e Jirau foram bem representadas pelos modelos lineares AR e ARMA, uma vez que este obteve o segundo menor EQM final (1056,3 $(m^3/s)^2$) no contexto da série de Água Vermelha, e aquele obteve o menor EQM final (393130 $(m^3/s)^2$) no contexto da série de Jirau. Já a flexibilidade adicional dos modelos não-lineares acabou dificultando o aprendizado efetivo dos comportamentos das séries, com exceção da RNN simples, a qual apresentou um desempenho equiparável ao dos modelos lineares: o menor EQM final (1048 ± 110 $(m^3/s)^2$) no contexto da série de Água Vermelha e o segundo menor EQM final (414914 ± 11516 $(m^3/s)^2$) no contexto da série de Jirau.

4 Conclusões

Os avanços da inteligência artificial (AI, em inglês *Artificial Intelligence*) nos últimos anos permitiu a criação de veículos autônomos (AVs, em inglês *Autonomous Vehicles*); de

programas que dominam jogos (*e.g.*, xadrez, GO), que reconhecem e classificam expressões faciais, voz e escrita, que realizam traduções instantâneas, e que propõe diagnósticos médicos [12, 13], a maior parte deles com capacidade sobre humana.

É possível que métodos de ML aplicados à previsão de séries temporais alcancem um desempenho igualmente sublime e superior ao dos modelos lineares. Contudo, a implementação desses métodos é mais difícil ~~do que aquelas citadas anteriormente~~, pois as regras para a predição de uma série são desconhecidas e mutáveis, existem instabilidades estruturais nos dados considerados e há aplicações em que as próprias predições podem alterar o futuro, aumentando ainda mais a incerteza das estimativas. Logo, pesquisas adicionais que experimentem ~~com ideias~~ inovadoras e criem ajustes aos modelos de previsão são necessárias para ultrapassar a capacidade de extraploração dos métodos estatísticos.

Neste contexto, introduzir o aluno a variadas estruturas de redes neurais, algumas com apelo moderno, o põe em contato com o cenário atual do problema de previsão de séries por aprendizado de máquina, o qual se encontra ainda em desenvolvimento e possui ampla possibilidade de expansão, seja na área da estatística, econometria, matemática financeira, ou ainda ambiental. Além disso, este projeto fornece os meios de compreensão básicos das diversas aplicações de redes neurais (por exemplo aquelas citadas anteriormente) e dá a ele a oportunidade de atuar, um dia, em uma dessas áreas.

5 Apêndice A - *Backpropagation*

O algoritmo de *backpropragation* tem como objetivo minimizar o erro cometido pela MLP. Sua dedução matemática será baseada no otimizador gradiente descendente, cuja equação está explícita em (5).

Escolheremos aleatoriamente o i-ésimo (de um total de I) peso $w_{i,j}^M$ pertencendo ao j-ésimo (de um total de J) neurônio da M-ésima (última) camada (a camada de saída) e calcularemos o gradiente da função de perda em relação a ele, aplicando a regra da cadeia:

$$\begin{gathered}\frac{\partial E}{\partial w_{i,j}^M} = \frac{\partial E}{\partial z_j^M}\frac{\partial z_j^M}{\partial w_{i,j}^M} = \delta_j^M \frac{\partial}{\partial w_{i,j}^M}\left(\sum_{k=1}^{I} w_{k,j}^M y_k^{M-1} + b_j^M\right) = \delta_j^M y_i^{M-1} \\ \text{com } \frac{\partial E}{\partial z_j^M} = \delta_j^M = \frac{\partial E}{\partial y_j^M}\frac{\partial y_j^M}{\partial z_j^M} = \frac{\partial E}{\partial y_j^M}\phi'(z_j^M)\end{gathered} \tag{11}$$

onde z_j^M é o resultado da soma ponderada do j-ésimo neurônio da M-ésima camada.

Temos como resultado das equações em (11) os seguintes gradientes para os pesos da última camada:

$$\begin{aligned}\frac{\partial E}{\partial w_{i,j}^M} &= \frac{\partial E}{\partial y_j^M}\phi'(z_j^M)y_i^{M-1} \\ \frac{\partial E}{\partial b_j^M} &= \delta_j^M = \frac{\partial E}{\partial y_j^M}\phi'(z_j^M)\end{aligned} \tag{12}$$

Note que a derivada da função de ativação $\phi'(z_j^M)$ calculada no ponto de interesse e a saída y_i^{M-1} do i-ésimo neurônio da *(M-1)*-ésima camada são grandezas que podem ser calculadas simplesmente a partir das saídas produzidas na etapa de propagação adiante, independentemente do neurônio da rede escolhido.

Já a derivada parcial da função de perda em relação à saída do j-ésimo neurônio da M-ésima camada, $\frac{\partial E}{\partial y_j^M}$, pode apenas ser calculada diretamente para a última camada. Isso, pois a função de perda E pode somente ser expressa diretamente em função das saídas da última camada.

Calcularemos, então, o gradiente de um dos pesos pertencentes ao k-ésimo neurônio da *(M-1)*-ésima camada. Primeiramente, note que a saída y_k^{M-1} deste neurônio interfere em todos os neurônios da próxima camada, e, portanto, ao considerarmos a derivada parcial da função de perda em relação a ela, devemos considerar na verdade a soma de erros que y_k^{M-1} propaga para a próxima camada (daí o nome). Deste modo:

$$\begin{aligned}\frac{\partial E}{\partial y_k^{M-1}} &= \sum_{j=1}^{J}\frac{\partial E}{\partial z_j^M}\frac{\partial z_j^M}{\partial y_k^{M-1}} \\ \text{com } \frac{\partial z_j^M}{\partial y_k^{M-1}} &= \frac{\partial}{\partial y_k^{M-1}}(\sum_{i=1}^{I} w_{i,j}^M y_i^{M-1} + b_j^M) = w_{k,j}^M\end{aligned} \tag{13}$$

Isto nos leva a:

$$\frac{\partial E}{\partial y_k^{M-1}} = \sum_{j=1}^{J}\delta_j^M w_{k,j}^M \tag{14}$$

sendo δ_j^M o mesmo que anteriormente e $w_{k,j}^M$ o peso que conecta o j-ésimo neurônio da M-ésima camada à k-ésima saída da *(M-1)*-ésima camada, ou seja, y_k^{M-1}.

Agora, definiremos a grandeza δ_i^{M-1} do mesmo jeito que anteriormente, porém com respeito a *(M-1)*-ésima camada:

$$\delta_i^{M-1} = \frac{\partial E}{\partial z_i^{M-1}} = \frac{\partial E}{\partial y_i^{M-1}}\frac{\partial y_i^{M-1}}{\partial z_i^{M-1}} = (\sum_{j=1}^{J}\delta_j^M w_{k,j}^M)\phi'(z_i^{M-1}) \tag{15}$$

Temos, por fim, o gradiente de erro de um dos pesos pertencentes ao k-ésimo neurônio da *(M-1)*-ésima camada:

$$\frac{\partial E}{\partial w_{l,i}^{M-1}} = \frac{\partial E}{\partial z_i^{M-1}} \frac{\partial z_i^{M-1}}{\partial w_{l,i}^{M-1}} = \delta_i^{M-1} \frac{\partial}{\partial w_{l,i}^{M-1}} (\sum_{k=1}^{L} w_{k,i}^{M-1} y_k^{M-2} + b_i^{M-1}) = \delta_i^{M-1} y_l^{M-2}$$
$$\text{e} \quad \frac{\partial E}{\partial b_i^{M-1}} = \delta_i^{M-1} \tag{16}$$

e novamente, este gradiente equivale à multiplicação do δ relativo ao neurônio que contém o peso em questão e a saída y da camada anterior que é ponderada por este peso. É possível, utilizando-se a relação entre os δs de camadas consecutivas estabelecida na equação (15) encontrar os valores de todos os δs de todos os neurônios da rede, começando pelos da última camada.

Fazendo isso, calcula-se a partir das equações em (16) o gradiente de qualquer um dos pesos, incluindo o bias, da rede. Os pesos são então atualizados de acordo com a expressão (5). Relembrando, é claro, que no caso da divisão dos dados em *mini-batches*, o gradiente utilizado na atualização de um peso é a média dos d gradientes em relação aquele peso calculados a partir de um *mini-batch*.

Referências

[1] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*, 5th ed. Wiley, 2015.

[2] L. Boccato, "Aplicação de computação natural ao problema de estimação de direção de chegada," Dissertação de Mestrado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, 2010.

[3] D. E. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization," *2nd Int. Conf. on Genetic Algorithms*, 1987.

[4] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd ed. O'Reilly Media, Inc., 2019.

[5] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[6] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv:1412.3555 [cs.NE]*, 2014.

[7] L. Boccato, "Novas propostas e aplicações de redes neurais com estados de eco," Tese de doutorado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, 2013.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[9] S. Haykin, *Adaptive filter theory*, 5th ed. Prentice Hall, 2013.

[10] C. M. Bishop, *Pattern Recognition and Machine Learning*. Cambridge, United Kingdom: Springer, 2011.

[11] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "Statistical and machine learning forecasting methods: Concerns and ways forward," *PLOS ONE*, vol. 13, no. 3, pp. 1–26, 03 2018. [Online]. Available: https://doi.org/10.1371/journal.pone.0194889

[12] H. Liang, B. Y. Tsui, and H. Ni, "Evaluation and accurate diagnoses of pediatric diseases using artificial intelligence," *Nat Med*, vol. 25, pp. 433–438, 2019. [Online]. Available: https://doi.org/10.1038/s41591-018-0335-9

[13] D. Ardila, A. P. Kiraly, and S. Bharadwaj, "End-to-end lung cancer screening with three-dimensional deep learning on low-dose chest computed tomography," *Nat Med*, vol. 25, pp. 954–961, 2019. [Online]. Available: https://doi.org/10.1038/s41591-019-0447-x